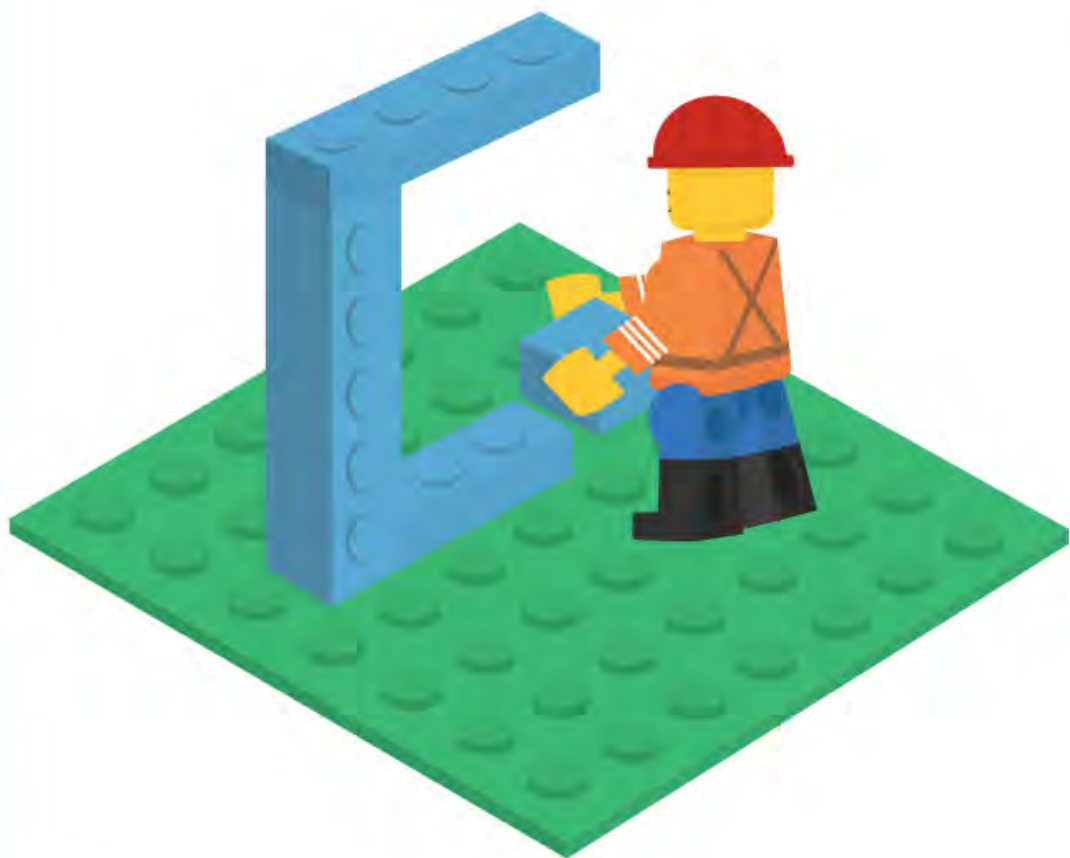
Introdução à programação em C

Os primeiros passos de um desenvolvedor

MAURÍCIO ANICHE

Quem sou eu?

Meu nome é Maurício Aniche, e trabalho com desenvolvimento de software pelos últimos 10 anos. Em boa parte desse tempo, atuei como consultor para diferentes empresas do mercado brasileiro e internacional. Com certeza, as linguagens mais utilizadas por mim ao longo da minha carreira foram Java, C# e C.

Como sempre pulei de projeto em projeto (e, por consequência, de tecnologia em tecnologia), nunca fui a fundo em nenhuma delas. Pelo contrário, sempre foquei em entender princípios que pudessem ser levados de uma para outra, para que, no fim, o código saísse com qualidade, independente da tecnologia.

Na academia, formei-me em Ciência da Computação pela Universidade Presbiteriana Mackenzie, em 2007. Em seguida, fiz mestrado em Ciência da Computação pela Universidade de São Paulo, onde defendi minha dissertação em 2012. Atualmente, sou aluno de doutorado pelo mesmo instituto.

Prefiro dizer que sou uma pessoa que tem um pé na academia e outro na indústria. Gosto dos dois mundos. Atualmente, trabalho pela Caelum, como consultor e instrutor. Também sou responsável pelos cursos do Alura, o portal de ensino à distância da Caelum.

Ensinar, sem dúvida, é o que mais gosto de fazer.

Agradecimentos

Agradecer é sempre a parte mais difícil. Em primeiro lugar, agradeço a meus pais por sempre apoiarem toda e qualquer iniciativa que tenho. Ver o quanto eles ficam orgulhosos a cada pequena conquista em minha carreira me faz querer sempre mais e mais.

Agradeço também ao meu amigo e colega de trabalho, Guilherme Silveira, pelas incontáveis discussões que tivemos ao longo da escrita deste livro. A versão Ruby foi escrita por ele, e segue as mesmas ideias deste. Criamos juntos toda a linha de ensino desses livros.

Por fim, a todos meus amigos, que me aguentam falando o tempo todo sobre o quanto é difícil e trabalhoso escrever um livro.

Forte abraço a todos!

Sumário

Capítulo 1

Introdução

Você já programou alguma vez na vida? Pois então, prepare-se: sua vida vai mudar. Ensinar a máquina a fazer o que quer que ela faça é extremamente divertido. E, por sorte, também útil.

Porém, o caminho é longo. Você precisará aprender a pensar como uma máquina, que não sabe e não faz nada que você não a mandar fazer, e a saber expressar-se em uma linguagem que ela entende.

Aqui, optamos pela linguagem C. C é uma linguagem bastante popular, por vários motivos diferentes. Ela é usada há muito tempo, por programadores do mundo inteiro, nos mais diferentes projetos. É extremamente poderosa: sistemas operacionais são feitos em C, por exemplo; e simples: sua sintaxe é bem fácil de ser assimilada.

C é também uma ótima linguagem para aqueles que querem aprender programação, pois ela nos dá diferentes níveis de abstração. Podemos es-

quecer que a memória do computador existe, ou manipulá-la diretamente, se quisermos.

Você provavelmente já passou por outros livros de programação. Eles não me agradam por alguns motivos. Primeiro, porque a maioria deles ensina o aluno a programar usando exemplos matemáticos. Cá entre nós, a matemática é legal, mas não queremos vê-la o tempo inteiro. Existem exemplos melhores, com certeza. Outros fazem projetos extremamente simples, como uma pequena calculadora ou uma pequena agenda. Mas quem quer, ao final de um livro enorme, ter uma simples agenda feita?

Minha proposta para este livro é ensinar você a programar por meio de jogos. Escrever um jogo é extremamente desafiador. Eles são cheios de lógicas complicadas e casos excepcionais. Um dos jogos, por exemplo, é similar ao Pac-Man, que você provavelmente jogou quando criança. Fazer o herói andar por um mapa, fugindo dos fantasmas, matando ou sendo mortos por eles, e comendo pílulas de energia é com certeza muito mais difícil de ser feito do que uma simples agenda.

Mas calma, não começaremos com o Pac-Man. Ao longo deste livro, desenvolveremos 3 grandes jogos: um **jogo de adivinhação**, no qual o computador pensará em um número e você chutará até acertar. Ele dará dicas se o número é maior ou menor do qual você chutou e, claro, você terá poucas chances para adivinhar. Perceba que, apesar de simples, temos muitos problemas computacionais para resolver: condicionais, loops, números randômicos, e assim por diante.

Depois, iremos para um **jogo de forca**. O computador escolherá uma palavra de uma lista, e desenhará a forca. O jogador, então, chuta letras e o computador vai mostrando os acertos, ou desenhando o boneco a cada erro. Os desafios aqui também são grandes: arrays, leitura e escrita de arquivos, funções separadas para que haja reúso entre elas, ponteiros de memória, e mais loops e condicionais.

Por fim, o **Foge-foge**, similar ao Pac-Man. Como já explicado, o herói foge dos fantasmas, que tentam pegá-lo. Aqui, aprendemos mais sobre matrizes/arrays bidimensionais, criação de estruturas e tipos próprios, e funções recursivas e diretivas de compilação.

Esses jogos nos darão muitas oportunidades para discussões e muito có-

digo. Esses códigos, aliás, são feitos passo a passo. Programadores erram, mudam de ideia e os melhoram aos poucos, conforme vão passando pelos desafios do programa que querem fazer. Aqui você verá que isso acontece. Nós mudaremos de ideia muitas vezes sobre o código que temos. Descobriremos que algo que fizemos no passado não era na verdade tão bom assim, e melhoraremos. É assim que programadores aprendem de verdade: vendo um código crescer do zero.

Enfim, ao final da leitura, você entenderá tudo o que precisa para começar a criar programas de computador.

Ah, esqueci de dois detalhes importantes!

Primeiro: jogos são legais! ;-)

Segundo: Se você tiver qualquer dúvida durante a leitura, venha conversar comigo na lista de discussão do livro, disponível em https://groups.google.com/forum/#!forum/introducao-a-programacao-em-c. Vamos lá bater um papo sobre programação!

1.1 A quem se destina este livro?

Este livro destina-se àqueles que querem aprender a programar. Não há pré-requisitos. Você não precisa conhecer alguma linguagem de programação ou mesmo ter uma lógica de programação bem afiada.

Este deve ser o seu primeiro livro de programação. Aquele que você guardará com carinho.

1.2 Como devo lê-lo?

Idealmente, você deve ler os capítulos na sequência. Ao longo deles, vamos evoluindo cada um dos jogos e misturando teoria com a prática. Pular um capítulo pode ser problemático, e o seu código pode não funcionar, se você perder algo.

Entretanto, ao final de cada capítulo, damos um link para o código completo até aquele momento. Se você estiver perdido ou achar que pode pular aquela discussão, nada o impede de baixar o código e ir adiante. Mas garanto a você que há surpresas e discussões interessantes em todos eles.

Capítulo 2

Jogo de adivinhação

Ao longo dos próximos capítulos, desenvolveremos um jogo de adivinhação. O computador pensará em um número, e você, jogador, precisará adivinhá-lo.

A cada erro, a máquina lhe dirá se o número chutado foi maior ou menor do que o pensado. Você também poderá escolher o nível de dificuldade do jogo, e isso lhe dará mais ou menos oportunidades de chutar um número. Ao final, se você ganhar, o computador lhe dirá quantos pontos você fez, baseando-se em quão bons eram seus chutes.

O jogo parece simples, mas ele lhe ajudará a aprender diversos conceitos importantes sobre programação, como:

- Ler do teclado e escrever na tela. Afinal, precisamos interagir com o usuário;
- Armazenar valores na memória e manipulá-los;
- Executar operações matemáticas em cima dessas variáveis;
- Entender os diferentes tipos de variáveis, e qual usar para guardar números inteiros e números com ponto flutuante;
- Tomar decisões no programa, baseando-se na entrada do usuário;
- Usar loops para executar o mesmo trecho de código repetidas vezes.

2.1 Como ele ficará?

Veja só como nosso jogo ficará ao final destes capítulos. Ao começar, ele pedirá para que o usuário escolha um nível de dificuldade:

.

```
Bem-vindo ao
Jogo de Adivinhação!
```

```
Qual o nível de dificuldade?
(1) Fácil (2) Médio (3) Difícil

Escolha:
```

Depois, ele começará a pedir palpites para o usuário. Enquanto o usuário errar, ele continuará pedindo.

```
-> Tentativa 1 de 6
Chute um número: 5

Seu chute foi menor do que o número secreto!

-> Tentativa 2 de 6
Chute um número: 6

Seu chute foi menor do que o número secreto!

-> Tentativa 3 de 6
Chute um número: 7

Seu chute foi menor do que o número secreto!

-> Tentativa 4 de 6
Chute um número: 8

Seu chute foi menor do que o número secreto!

-> Tentativa 5 de 6
Chute um número: 9

Seu chute foi menor do que o número secreto!

-> Tentativa 6 de 6
Chute um número: 10

Seu chute foi menor do que o número secreto!
```

Se você esgotar todas suas chances, ele lhe avisará que você perdeu:

```
.

       \|/ ____ \|/
        @~/ ,. \~@
       /_( \__/ )_\
          \__U_/

Você perdeu! Tente novamente!
```

Se você acertar, você saberá quantos pontos fez:

```
.
             00000000000
         0000000000000000000
      000000  000000000  000000
    000000      00000      000000
  00000000  #   00000  #   00000000
 0000000000    0000000    0000000000
0000000000000000000000000000000000000
0000000000000000000000000000000000000
0000  0000000000000000000000000  0000
 0000  00000000000000000000000  0000
  0000   00000000000000000000  0000
    00000   000000000000000   0000
      000000   000000000   000000
         000000         000000
             000000000000

Parabéns! Você acertou!
Você fez 985.00 pontos. Até a próxima!
```

2.2 É hora de começar!

Está pronto para desenvolver esse jogo? **Mãos à obra!**

Capítulo 3

Variáveis

Estamos prontos para começar nosso jogo de adivinhação. Mas antes de entrarmos de cabeça nele, precisamos escolher uma linguagem de programação. Na verdade, você já sabe qual é, pois ela está na capa deste livro. Tudo o que escrevermos aqui será na linguagem C. É interessante que ela, apesar de ser uma linguagem antiga, ainda é bastante utilizada em muitos lugares importantes no nosso dia a dia. Sistemas operacionais, por exemplo, são escritos em C.

Por ser uma linguagem popular, muitas das mais modernas, como o caso do Java e do C#, foram baseadas nela. Ou seja, começar por C é uma boa escolha! Nesta parte do livro, nós estudaremos seus conceitos básicos e, com eles, implementaremos nosso primeiro jogo: o jogo da adivinhação.

3.1 Nosso primeiro programa

Todo programa em C começa por um arquivo que contém todo o código-fonte do nosso programa. Esse arquivo pode ser escrito em qualquer editor de textos simples, como o Bloco de Notas. É fácil saber que um arquivo é um programa em C, simplesmente porque ele termina com a extensão `.c`. O primeiro passo é criar um arquivo e salvá-lo como `adivinhacao.c` (*adivinhacao* é o nome do jogo). Geralmente, damos um nome significativo para esse arquivo, afinal quando estivermos desenvolvendo um sistema maior, teremos muitos deles. Também evite qualquer tipo de acento ou espaços em seu nome.

> **Editores de código**
>
> Existem muitos editores de código para programadores. Uma das IDEs (nome que damos a editores de texto focados em código) mais populares é o Eclipse. Ele é muito usado por programadores Java. Entretanto, nesse momento do seu aprendizado, sugiro a você usar uma IDE mais simples, que o ajude menos e o faça pensar mais no código que está escrevendo. Recomendo editores como o Sublime Text, Notepad++, Ultraedit, ou qualquer outro que somente deixe o código colorido, para facilitar a visualização.

Nesse arquivo, vamos começar já com a mensagem de boas-vindas do nosso jogo. Ao rodar, ele deverá dizer "Bem-vindo ao Jogo de Adivinhação". Todo texto deve ser escrito entre aspas; assim a linguagem C sabe que se trata de um texto qualquer e não de algum "comando especial" da linguagem. No arquivo, então:

```
"Bem-vindo ao Jogo de Adivinhação"
```

Porém, isso não é suficiente. Precisamos dizer ao programa para que imprima o texto na saída padrão, ou seja, no monitor. Para isso, usaremos nossa primeira **função**. Mais adiante, veremos com muito mais detalhes, mas nesse momento, uma função é um conjunto de código que faz uma tarefa bem defi-

nida. A linguagem C, vem com a função `printf` que, como o próprio nome diz, imprime. Para imprimirmos algo na tela, usando o `printf`, fazemos:

```
printf("Bem-vindo ao Jogo de Adivinhação")
```

Veja que abrimos parênteses e passamos o nosso texto entre aspas. Podemos chamar essa função quantas vezes quisermos, com o texto que quisermos, e ele sempre será impresso na tela. Veja só no código a seguir, onde utilizamos o `printf` várias vezes seguidas, para imprimir uma mensagem mais amigável e bonita para o usuário final.

Repare também no ponto e vírgula! Toda linha em C termina com ponto e vírgula. Parece estranho, mas você logo se acostumará!

```
printf("************************************");
printf("* Bem-vindo ao Jogo de Adivinhação *");
printf("************************************");
```

Estamos quase prontos para executar nosso programa. Esse monte de `printfs` não pode ficar jogado dentro do nosso arquivo C. Precisamos colocá-los dentro da "função principal" do programa. Todo programa em C tem uma função principal. É ela que é executada quando você roda o seu arquivo `.exe`, `.out`, ou qualquer que seja a extensão executável do seu sistema operacional. Essa função chama-se `main`.

Precisamos também dizer ao C que estamos usando a função `printf`. Afinal, ela precisou ser escrita em algum lugar, certo? Para isso, escreveremos no começo no nosso arquivo, `#include <stdio.h>`. É fácil entender `stdio`: vem de *standard I/O*, ou seja, entrada e saída padrão. É nesse arquivo que ficam todas as funções de entrada (ler do teclado, por exemplo), e saída (escrever no monitor). Você verá que todos nossos programas incluirão esse arquivo.

Nesse momento, não se preocupe com todo o código que aparecerá. Você vai entender cada parte dele mais à frente. Por enquanto, entenda que, ao rodar o programa, tudo que estará dentro desse `main` será executado. Repare também nas chaves { }. Elas delimitam o "corpo" dessa função. Ou seja, tudo que está dentro das chaves, está dentro dessa função.

```c
#include <stdio.h>

int main() {
    printf("************************************");
    printf("* Bem-vindo ao Jogo de Adivinhação *");
    printf("************************************");
}
```

Programa feito. Precisamos agora executá-lo. Entretanto, antes disso, queremos transformá-lo em código de máquina. O programa, escrito em C, é ótimo para outros seres humanos lerem e modificarem, mas péssimo para a máquina. Precisamos transformá-lo em código que a máquina entende, ou seja, em código de máquina. É para isso que serve um **compilador**. Ele pega o arquivo que está escrito na sua linguagem de programação e o transforma em código de máquina (que nós não conseguimos entender).

O compilador mais conhecido da linguagem C é o GCC. Se você está usando Linux ou Mac, ele já vem instalado. Se você está no Windows, precisará instalá-lo. Veja o apêndice do livro se tiver dúvidas de como fazer isso. Com o compilador em mãos, vamos pedir para ele gerar um executável. Para isso, vamos invocá-lo, usando o terminal, passando para ele o nome do arquivo `.c` e o nome do executável que queremos gerar (logo após o `-o`).

Acostume-se com o terminal (ou *Command Prompt*), pois o usaremos bastante ao longo do livro. Entre no diretório em que você salvou o código-fonte (se você ainda não sabe como usar o terminal, vá ao nosso apêndice), e execute o comando a seguir:

No Windows:

```
gcc adivinhacao.c -o adivinhacao.exe
```

No Linux ou Mac (a única diferença é que tiramos o `.exe` do nome do arquivo):

```
gcc adivinhacao.c -o adivinhacao
```

Vamos agora executar o programa. No Linux ou Mac, basta fazer `./adivinhacao`. No Windows, basta digitar `adivinhacao`.

Comemore! Nosso primeiro programa executou e nos imprimiu "Bem-vindo ao Jogo de Adivinhação".

Ainda não é o que queremos, afinal ele imprimiu tudo na mesma linha. O `printf` não quebra linha se você não mandar. Veja que a máquina não faz nada que você não queira explicitamente! Para que nosso programa dê *enter* entre uma linha e outra, basta acrescentarmos um `\n` ao final de cada `printf`. O `\n` é o código para *enter* em nossos programas em C. Veja como nosso código fica:

```
#include <stdio.h>

int main() {
    printf("************************************\n");
    printf("* Bem-vindo ao Jogo de Adivinhação *\n");
    printf("************************************\n");
}
```

Lembre-se de que, a cada mudança feita no código-fonte, você deve compilar novamente o programa. Se rodarmos, ele nos imprimirá da maneira como queremos. Parabéns, temos nosso primeiro programa rodando.

```
************************************
* Bem-vindo ao Jogo de Adivinhação *
************************************
```

Comentários

Podemos inserir comentários em nosso código; ou seja, trechos de texto em português, que serão ignorados pelo compilador. Eles nos ajudam a descrever melhor o que queremos fazer. Para fazer um comentário, basta começar a linha com `//`. Tudo o que vier na sequência será completamente ignorado pelo compilador. Por exemplo:

```
// comentário, isso não faz nada
printf("Oi");
// outro comentário, também não faz nada
```

Se você não estiver entendendo alguma parte do código, coloque um comentário. Mas também não abuse deles: um código com muitos fica difícil de ser lido.

3.2 Declarando variáveis

Nosso próximo passo agora é ter um número secreto, e começar a pedir para o usuário chutar qual é. Começaremos guardando esse número em algum lugar. Algo muito comum em nossos programas será guardar dados/ informações, para que possamos manipulá-las mais para frente. Esse número secreto, por exemplo, será algo que usaremos ao longo de todo nosso programa e, por isso, faz sentido guardá-lo em algum lugar fácil.

Chamamos esses lugares de **variáveis**. Para declararmos uma variável em C, precisamos de duas coisas: de um nome qualquer, pois é assim que nos referenciaremos a ela depois; e saber o tipo de dado que ela guardará. Nossos programas diferenciam tipos de dados. Ou seja, números inteiros são números inteiros, números com vírgula são números com vírgula, e textos (que a partir de agora chamarei de *string*) são strings. A partir do momento em que você disse que uma variável guarda um número inteiro, você só poderá colocar números inteiros nela.

Vamos chamar essa variável de `numerosecreto`, e ela será do tipo `int` (que significa inteiro). Em seguida, vamos colocar o número secreto, que, por enquanto será 42, dentro dela. Veja as duas novas linhas em nosso código,

uma declarando a variável, e a outra dando um valor a ela, usando o sinal de igual:

```
int main() {
    printf("************************************\n");
    printf("* Bem-vindo ao Jogo de Adivinhação *\n");
    printf("************************************\n");

    int numerosecreto;
    numerosecreto = 42;
}
```

Variável com nome repetido?

Variáveis devem ter nomes únicos dentro do escopo que estão. Ou seja, não podemos declarar duas variáveis `numerosecreto`. Se fizermos isso, por algum engano, o compilador reclamará e não nos gerará o executável do programa. A mensagem será algo como: *error: redefinition of 'numerosecreto'*. Ou seja, "redefinição" da variável.

Faça o teste. Crie uma variável com o mesmo nome e invoque o GCC.

Com essa variável em mãos, poderíamos, por exemplo, imprimi-la. Mas a pergunta é: como imprimimos o conteúdo de uma variável? Precisamos, de alguma forma, avisar ao `printf()` que queremos imprimir, por exemplo, a variável `numerosecreto`.

Uma primeira tentativa poderia ser algo como:

```
printf("O número numerosecreto é secreto!");
```

Mas, obviamente, o programa não trocará o texto `numerosecreto` pelo conteúdo da variável. Ele não tem como saber disso. Precisamos deixar uma "máscara" na string; algo que o `printf()` saiba que será substituído por uma variável: para isso, usamos o **%d**. Quando colocamos o `%d` no meio da string, significa que queremos que ali seja colocada uma variável do tipo

inteiro. Existem várias outras máscaras, que veremos ao longo do livro. Passamos também para a função `printf()` a variável `numerosecreto` que queremos imprimir.

Para isso, basta passarmos um segundo argumento para a função, separando por vírgula. Veja em código:

```
int numerosecreto;
numerosecreto = 42;

// imprimindo somente o número
printf("%d", 42);

// no meio da frase
printf("O número %d é secreto!");
```

Porém, claro, vamos jogar esse código fora, pois o jogador não pode saber o número secreto. Isso acabaria com a graça do jogo.

3.3 Lendo do teclado

Agora que já temos onde guardar o número secreto e a quantidade de jogadas, vamos pedir para o usuário digitar um número. De forma análoga ao `printf`, temos a função `scanf`, que lê do teclado. A função `scanf` precisa saber duas coisas: o tipo do dado que ela vai ler, e em qual variável ela deverá colocar o valor lido. Como o usuário digitará um número, declararemos uma variável do tipo `int` e falaremos para a função que é um número que será digitado: é isso que o `%d` indica, idêntico ao que usamos no `printf`:

```
int chute;

printf("Qual é o seu chute? ");
scanf("%d", &chute);
printf("Você chutou o número %d!", chute);
```

Leia cada uma dessas linhas com atenção, pois elas têm muitas novidades. Na primeira, declaramos a variável `chute`. Depois imprimimos uma mensagem na tela. Até aqui, você já conhece tudo. Em seguida, usamos o

`scanf` para ler a variável. Passamos `%d`, porque queremos ler um número, e queremos guardá-lo na variável `chute`. Repare no `&`: não se preocupe com ele por enquanto, mas guarde que você sempre deve usá-lo no `scanf`.

Logo após, imprimimos uma mensagem, e também usamos o `%d`. Veja que a função `printf` recebeu depois a variável `chute`. O que ela fará? Ela substituirá o `%d` pelo valor que está na variável `chute`. Ela é uma função bastante inteligente:

```
printf("Seu chute foi %d\n", chute);
```

Veja nosso programa rodando:

```
************************************
* Bem-vindo ao Jogo de Adivinhação *
************************************
Qual eh o seu chute? 12
Seu chute foi 12
```

Acostume-se com variáveis, afinal nós as usaremos o tempo inteiro. A maneira de declará-las será sempre a mesma: o tipo da variável e o nome dela. Com o nome, podemos ler ou mudar o seu valor. Todos os capítulos daqui para frente declararão variáveis dos mais diferentes tipos.

3.4 Entendendo o compilador

Sempre que quisermos rodar nosso programa em C, precisamos antes passá-lo pelo GCC. Mas o que acontece ali? O GCC é o que chamamos de **compilador**. Compilador é o programa que pega o código escrito em uma linguagem de programação como C, Java, C#, ou qualquer outra linguagem existente , e o transforma em uma "outra linguagem". Essa outra linguagem, no caso dos compiladores C, é a linguagem de máquina, que é entendida somente por aquele sistema operacional e arquitetura usada (por exemplo, Windows em máquinas de 32-bits, ou Linux em máquinas Sun Sparc).

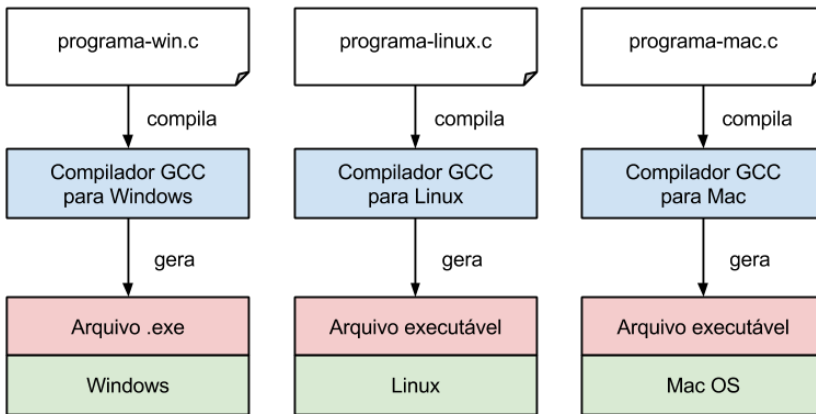



Fig. 3.1: Funcionamento de um compilador

O trabalho do compilador não é nada fácil. Afinal, ele precisa abrir o seu arquivo `.c`, e ver se tudo o que você escreveu obedece as regras da linguagem C (usamos até o termo *gramática* da linguagem C). Se você esqueceu um ponto e vírgula, por exemplo, o compilador precisa parar o processo e avisar o desenvolvedor. Depois, se tudo estiver de acordo com as regras gramáticas da linguagem, ele precisa pegar cada instrução dessa e converter para código de máquina.

Hoje em dia, compiladores são bastante inteligentes. Eles conseguem perceber trechos de código que podem ser otimizados, e muitos deles até modificam internamente o seu código para que a saída seja a melhor possível.

O grande problema de programas escritos em C é que eles não são portáveis. Ou seja, o executável gerado pelo seu compilador GCC no Windows não funcionará no Linux; o gerado no Linux não funcionará no Mac, e assim por diante. Se o seu programa em C só usou instruções padrões da linguagem, você conseguirá pegar o mesmo `.c` e compilar em ambas as plataformas, gerando dois executáveis diferentes; entretanto, muitos programas em C usam funções específicas de sistemas operacionais específicos, perdendo sua portabilidade. Esse é um problema que já foi bem resolvido por linguagens mais modernas, como Java e C#. O funcionamento deles não será discutido nesse

momento.

3.5 Resumindo

Neste capítulo, aprendemos:

- A escrever nosso primeiro programa em C, e a escrever a função `main`;
- A ler e a escrever da entrada e saída padrão do sistema, por meio de `printf` e `scanf`;
- Aprendemos também a declarar variáveis do tipo inteiro;
- Entendemos o papel do compilador em todo o processo.

Capítulo 4

Controle de fluxo com ifs e fors

O próximo passo do nosso jogo é dizer se o chute do usuário acertou ou não o número secreto. Se ele acertou, daremos parabéns; se errou, temos que avisá-lo. O código poderia ser algo como:

```
printf("Qual é o seu chute? ");
scanf("%d", &chute);
printf("Seu chute foi %d\n", chute);

printf("Parabens, você acertou!");
printf("Poxa vida, você errou!");
```

Porém, se rodarmos nosso código do jeito que está, o comportamento do programa será estranho: ele imprimirá ambas as mensagens. Para que isso fique certo, precisamos descobrir se podemos imprimi-las. Imprimiremos uma ou outra. Portanto, nosso programa precisa tomar uma decisão.

É isso que nossos programas fazem o tempo todo, e é justamente por isso que eles são úteis. Conseguimos ensiná-los a tomar decisões baseando-se nas informações que temos em mãos. Em nosso jogo, a decisão é fácil: se o chute dado pelo usuário for igual ao número secreto, avisaremos que ele acertou; caso contrário, avisaremos que errou.

Em pseudocódigo, é algo como:

```
SE chute FOR IGUAL A numero secreto, ENTAO
    avisa que ele acertou
CASO CONTRARIO
    avisa que ele errou
```

Em código, é bem parecido, mas usaremos palavras em inglês. Vamos começar com o "se", que em inglês, é `if`, e o "caso contrário", que é `else`. Agora entra a sintaxe do `if` na linguagem C. Veja que escrevemos `if` e, dentro dos parênteses, colocamos a condição que ele deve avaliar. Repare só na igualdade, que usamos dois iguais. Isso é para diferenciar do um único igual (usado para atribuir valores para variáveis).

Repare também no uso das chaves. Sempre que temos um `if`, abrimos uma chave pra indicar que ali vem um novo bloco de código. Esse bloco deverá ser executado, caso aquela condição seja verdadeira.

```
printf("Qual é o seu chute? ");
scanf("%d", &chute);
printf("Seu chute foi %d\n", chute);

if(chute == numerosecreto) {
    printf("Parabéns! Você acertou!");
} else {
    printf("Você errou!")
}
```

Repare que esse bloco de código é idêntico ao bloco que abrimos quando criamos a função `main()`. Podemos ter quanto código quisermos dentro dele. Veja o exemplo a seguir, onde colocamos mais código dentro dos blocos:

```
printf("Qual é o seu chute? ");
scanf("%d", &chute);
```

```
printf("Seu chute foi %d\n", chute);

if(chute == numerosecreto) {
    printf("Parabéns! Você acertou!\n");
    printf("Jogue de novo, você é um bom jogador!\n")

    // poderíamos ter mais código aqui
    // ...
} else {
    printf("Você errou!\n");
    printf("Mas não desanime, tente de novo!\n");

    // poderíamos ter mais código aqui
    // ...
}
```

Mas o usuário precisa de mais dicas que isso. Precisamos dizer se o chute que ele deu foi maior, menor ou se acertou em cheio. Nosso programa precisa fazer mais comparações. O chute é maior que o número secreto? O chute é menor que o número secreto? O chute é igual ao número secreto?

Em que momento devemos fazer isso? Devemos verificar se ele é maior e menor dentro do `else`. Afinal, é aquele trecho de código que será executado se o chute foi diferente do número secreto. Podemos, sem qualquer problema, colocar `ifs` dentro de `ifs`. Nossa condição agora será um pouco diferente. Olharemos se o chute é maior (>) ou menor (<) que o número secreto. Veja o código:

```
if(chute == numerosecreto) {
    printf("Parabéns! Você acertou!\n");
} else {
    if(chute > numerosecreto) {
        printf("Seu chute foi maior do que o número secreto!\n");
    }
    if(chute < numerosecreto) {
        printf("Seu chute foi menor do que o número secreto!\n");
    }

}
```

Teste seu programa!

Acostume-se desde já a testar o seu programa de computador. Até agora, nosso programa tem 3 caminhos diferentes: o caso de o jogador acertar o número, o caso de o número ser menor e o caso de ser maior.

Rode o programa, no mínimo, 3 vezes, e teste cada uma das possibilidades. É nossa responsabilidade garantir que tudo funciona como esperado.

Aprenda desde já que um *bug*, ou seja, um defeito no programa, pode custar muito dinheiro aos nossos usuários finais. De novo, **teste seus programas**.

Se rodarmos nosso programa agora, ele imprimirá a mensagem certa, de acordo com o chute do usuário. Sempre que temos nosso código funcionando, é hora de melhorar sua legibilidade. Condições dentro de `ifs` geralmente são difíceis de serem lidas e entendidas. Por exemplo, apesar de esse `chute == numerosecreto` ser simples, em um sistema maior, essa condição poderia ser maior. Uma sugestão para isso é colocar essa condição dentro de uma outra variável, com um nome melhor, que explique aquela condição. Por exemplo, esse `if` trata o caso de o usuário ter acertado o número secreto. Vamos colocá-lo em uma variável cujo nome deixe isso claro:

```
int acertou = chute == numerosecreto;

if(acertou) {
    printf("Parabéns! Você acertou!\n");
} else {
    if(chute > numerosecreto) {
        printf("Seu chute foi maior do que o número secreto!\n");
    }
    if(chute < numerosecreto) {
        printf("Seu chute foi menor do que o número secreto!\n");
    }
}
```

Veja só que a variável `acertou` tem um nome muito melhor e deixa claro o que está acontecendo. Repare que ela é do tipo inteiro. A linguagem C não tem um tipo para guardar verdadeiros ou falsos (outras linguagens têm o tipo `boolean`, que recebe apenas `true` ou `false`).

Mas qual o valor que está dentro da variável? A convenção da linguagem é que `0` significa "falso", ou seja, quando a expressão avaliada não for verdadeira. E `1` é o contrário: significa verdadeiro. Podemos até imprimir a variável, rodar nosso programa algumas vezes e ver os valores que estão nela:

```
int acertou = chute == numerosecreto;

// imprimirá 0 quando a condição for falsa,
// e 1 quando ela for verdadeira.
printf("Acertou: %d\n", acertou);

if(acertou) {
    printf("Parabéns! Você acertou!\n");
} else {
    if(chute > numerosecreto) {
        printf("Seu chute foi maior do que o número secreto!\n");
    }
    if(chute < numerosecreto) {
        printf("Seu chute foi menor do que o número secreto!\n");
    }
}
```

Podemos fazer a mesma coisa para o `if` de dentro, e criar uma variável que explique o que está acontecendo. Dessa vez, não precisamos repetir o `if`; afinal, se os números são diferentes (pois entrou no `else`) e não é maior, logo será menor. Veja:

```
int acertou = chute == numerosecreto;

if(acertou) {
    printf("Parabéns! Você acertou!\n");
} else {
    int maior = chute > numerosecreto;
    if(maior) {
```

```
        printf("Seu chute foi maior do que o número secreto!\n");
    } else {
        printf("Seu chute foi menor do que o número secreto!\n");
    }
}
```

Tudo continua funcionando e nosso código está mais legível. Porém, ainda temos um problema: o usuário só consegue jogar uma vez.

4.1 Escopo de variáveis

Você percebeu que declaramos variáveis em vários pontos diferentes do nosso código? Temos a variável `chute` por exemplo, que foi declarada logo no começo da função `main`, bem como a variável `maior`, que foi declarada lá dentro do `else`.

Será que conseguimos acessar qualquer variável a qualquer momento? Por exemplo, será que conseguimos imprimir a variável `acertou` dentro do segundo `if`?

```
int acertou = chute == numerosecreto;

if(acertou) {
    printf("Parabéns! Você acertou!\n");
} else {

    printf("Acertou: %d\n", acertou);

    int maior = chute > numerosecreto;
    if(maior) {
        printf("Seu chute foi maior do que o número secreto!\n");
    } else {
        printf("Seu chute foi menor do que o número secreto!\n");
    }
}
```

Sim. Isso funciona. Sempre que declaramos uma variável, ela é disponível no **escopo** em que ela está e nos escopos mais adentro. Mas o que é **escopo**?

Escopo é o nome que damos para os trechos do programa, onde determinada variável é válida.

Declaramos a variável `chute` dentro do escopo da função `main`. Portanto, ela é válida dentro de toda a função e também nos escopos que são declarados dentro dela. Por exemplo, os `ifs` e `elses` que temos em nosso programa.

No entanto, a variável `maior` declarada dentro do `else` só é visível dentro do escopo desse `else` (delimitado pelas chaves que abrimos ao escrevê-lo) e nos escopos de dentro. Isso quer dizer que, se tentarmos usar essa variável em um escopo *maior* do que o do `else`, não conseguiremos.

```
int acertou = chute == numerosecreto;

printf("%d", maior);

// lembre-se de que o maior está declarado mais abaixo
// do código, dentro do else
```

Veja o erro que o compilador nos dá, se tentarmos usar essa variável fora do `else`:

```
error: use of undeclared identifier 'maior'
    printf("%d", maior);
                 ^
```

Logo, lembre-se de sempre declarar suas variáveis no escopo correto. Na prática, você sempre a declara no escopo *mais alto* que vai utilizá-la.

4.2 Loops e for

O usuário pode ter 3 chances para acertar o número. Ou seja, precisamos repetir o mesmo trecho de código várias vezes. Imagine como seria seu arquivo:

```
printf("Qual é o seu chute? ");
scanf("%d", &chute);
printf("Seu chute foi %d\n", chute);

if (...)
```

```
printf("Qual é o seu chute? ");
scanf("%d", &chute);
printf("Seu chute foi %d\n", chute);

if (...)

printf("Qual é o seu chute? ");
scanf("%d", &chute);
printf("Seu chute foi %d\n", chute);

if (...)
```

Aqui o arquivo está resumido. Agora imagine algo pior, você decidiu mudar a regra para 10 tentativas. Imagine copiar e colar 10 vezes o mesmo trecho de código? Agora imagine que decidamos alterar a frase *Qual é o seu chute?*. Precisaremos alterar em 10 lugares diferentes. É muito trabalho!

Desde já, perceba que código repetido é errado, e nos dá mais trabalho do que outra coisa. Muitos sistemas de verdade hoje sofrem com isso. Imagine trechos de código repetidos em diferentes arquivos, e toda vez que a regra de negócios muda, o desenvolvedor precisa encontrar todas essas duplicações. Quando ele esquece, um bug acontece. E você, que provavelmente é usuário de muitos softwares, já viu vários deles.

Podemos dizer ao nosso programa para repetir trechos de código, de forma automática. Para executar o mesmo código várias vezes, fazemos o que chamamos de **loops**. A primeira maneira de se escrever um loop é usando o `for`. Vamos usá-lo, pois sabemos exatamente a quantidade de vezes que queremos repetir um código; nesse caso, 3.

Um `for` contém três partes: a primeira é a variável que será usada de contador e o primeiro valor dela; a segunda é a condição de parada do loop; e o terceiro é o incremento.

Em português, parece complicado, mas a ideia é que se fôssemos fazer um loop de 10 a 30, precisaríamos: inicializar nossa variável com 10; dizer que a condição é que o número precisa ser menor de 30 para o loop acontecer; e que, a cada vez que o loop rodar, incrementaríamos nossa variável contador em 1 unidade.

Veja, em código, nosso loop, de 1 a 3. Inicializamos a variável `i` com o número 1, e fizemos a condição de parada como sendo `i <=3`. Como o incremento é um `i++`, ou seja, soma 1 à variável `i`, então `i` terá os valores 0, 1 e 2. Quando for igual a 3, a condição será falsa, e o loop parará:

```
for(int i = 1; i <= 3; i++) {
    printf("Qual é o seu chute? ");
    scanf("%d", &chute);
    printf("Seu chute foi %d\n", chute);

    int acertou = chute == numerosecreto;

    if(acertou) {
        printf("Parabéns! Você acertou!\n");
    } else {
        int maior = chute > numerosecreto;
        if(maior) {
            printf("Seu chute foi maior do que o número
                    secreto!\n");
        } else {
            printf("Seu chute foi menor do que o número
                    secreto!\n");
        }
    }
}

printf("Fim de jogo!\n");
```

Vamos entender passo a passo o que acontece com esse trecho de código:

1) O `for` começa.

2) Ele executa o primeiro bloco (bloco de inicialização), declara a variável `i` e atribui o valor 1 a ela.

3) Em seguida, ele valida a condição, que está no bloco 2. Como 1 é menor-ou-igual a 3, a comparação retorna verdadeira.

4) Como a condição é verdadeira, ele executa o bloco de código.

5) Em seguida, ele executa o terceiro bloco do `for`, que é o incremento. Ali, fazemos `i++`, e a variável passa a valer 2.

6) Novamente, ele valida a condição. Como 2 é menor-ou-igual a 3, a comparação retorna verdadeira.

7) Como a condição é verdadeira, ele executa o bloco de código.

8) Novamente, ele executa o incremento. A variável `i` passa a valer 3.

9) Ele valida a condição. Como 3 é menor-ou-igual a 3, a comparação retorna verdadeira.

10) Como a condição é verdadeira, ele executa o bloco de código.

11) Ele executa o incremento. A variável `i` passa a valer 4.

12) Ele valida a condição. Como 4 **não é** menor-ou-igual a 3, a comparação retorna falsa, e o loop acaba.

13) O programa continua e a mensagem "Fim de jogo" é impressa.

Portanto, para entender o funcionamento do `for`, basta entender os seus 3 blocos e quando cada um deles é executado. O primeiro bloco, o de inicialização, é executado uma única vez no começo. Em seguida, ele executa o segundo bloco, a condição. Se ela for verdadeira, ele dispara o bloco de código. Por fim, ele executa o terceiro bloco, o incremento. E aí repete, valida a condição, executa bloco, executa incremento, assim por diante, até a condição deixar de ser verdadeira.

Podemos até aproveitar essa variável `i` e exibir o número da tentativa. Assim, o jogador fica sabendo quantos chutes ele já deu. E isso é fácil, pois basta usarmos a variável no `printf`. Veja só o segundo `printf`. Repare que essa função pode receber mais de um parâmetro. É assim que fazemos quando temos várias variáveis para exibir na mesma linha:

```
for(int i = 1; i <= 3; i++) {
    printf("Qual é o seu %do. chute? ", i);
    scanf("%d", &chute);
    printf("Seu %do. chute foi %d\n", i, chute);
```

```
    // ...
}
```

Um detalhe interessante é pensar no escopo da variável `i`. Como ela foi declarada direto no `for` (veja que até usamos a sintaxe de declaração de variáveis: `int i = 1`), ela só existe dentro do `for`. Ou seja, se tentarmos imprimir essa variável ali junto com a mensagem de fim de jogo, por exemplo, tomaremos o erro do compilador, que dirá que a variável não existe.

4.3 Parando loops

Nosso jogo está indo muito bem. Já pedimos para o usuário chutar um número, e fazemos isso 3 vezes, de maneira automática. E ele também nos avisa se este é maior ou menor do que o número secreto. O problema é que, se acertarmos o número, ele continua nos pedindo mais números para chutar. E não deveria: o jogo deveria terminar ali. Veja:

```
************************************
* Bem-vindo ao Jogo de Adivinhação *
************************************
Qual é o seu chute? 42
Seu 1o. chute foi 42
Parabéns! Você acertou!
Qual é o seu chute?    <-- essa linha não deveria estar aqui
```

Apesar de termos feito nosso loop só parar quando `i` for menor que 3, temos um caso especial, em que precisamos parar o loop. É para isso que usamos a palavra `break`. Ela simplesmente para a execução do loop, e passa para a próxima linha, logo após o fim do laço. Precisamos quebrá-lo no momento em que o jogador acerta o número; ou seja, dentro do `if` que verifica isso. Vamos também colocar uma linha logo após o loop, para entendermos melhor o que acontecerá.

Veja o código:

```
for(int i = 1; i <= 3; i++) {
    printf("Qual é o seu %do. chute? ", i);
```

```
        scanf("%d", &chute);
        printf("Seu %do. chute foi %d\n", i, chute);

        int acertou = chute == numerosecreto;

        if(acertou) {
            printf("Parabéns! Você acertou!\n");
            break;
        } else {
            int maior = chute > numerosecreto;
            if(maior) {
                printf("Seu chute foi maior do que o número
                        secreto!\n");
            } else {
                printf("Seu chute foi menor do que o número
                        secreto!\n");
            }
        }
    }

    printf("Obrigado por jogar!");
```

Agora, ao jogarmos, ele continuará pedindo chutes, enquanto o jogador não acertar. Mas, se ele acertar, o programa para na hora, independente da jogada em que ele está:

```
************************************
* Bem-vindo ao Jogo de Adivinhação *
************************************
Qual é o seu 1o. chute? 10
Seu 1o. chute foi 10
Seu chute foi menor do que o número secreto!
Qual é o seu 2o. chute? 42
Seu 2o. chute foi 42
Parabéns! Você acertou!
Obrigado por jogar!
```

Nosso jogo está cada vez melhor.

4.4 Defines, constantes e números mágicos

Definimos que o jogador terá 3 chances para adivinhar o número secreto; e esse 3 está explícito em nosso código, ali dentro do `for`. Se quisermos dar 5 chances, por exemplo, bastaria trocar esse número. Repare que ele não é "simplesmente um número". Ele tem um significado importante para nosso código. É o número máximo de tentativas que um jogador pode ter.

Uma coisa importante sempre que pensamos em escrever programas de computador é escrever códigos que sejam fáceis de ser lidos por outros seres humanos. Afinal, raramente um programa *termina*. Ele geralmente é mantido por muitos e muitos anos e, se não for fácil entender o que está escrito ali, o custo de manutenção sobe. Uma das maneiras para garantir que o código seja fácil de ser mantido é dar bons nomes para tudo e evitar "números mágicos".

Números mágicos são aqueles que têm um significado importante para o código, e estão lá jogados. É o caso do nosso `3`. Se você mostrar esse código para outra pessoa, ele precisará lê-lo inteiro para entender o que significa; ao passo que, se trocarmos o número 3 pela palavra `NUMERO_DE_TENTATIVAS`, tudo fica mais claro.

Quando temos números fixos (constantes, que não variam) e importantes em nosso sistema, geralmente damos um nome a ele, igual fizemos anteriormente. Em C, podemos usar a diretiva `#define`. Ela nos ajuda a definir uma constante; ou seja, um valor que nunca mudará ao longo do código. Portanto, em vez de usar o número mágico, usamos o nome definido. Veja sua definição logo abaixo do `#include`. Repare que não usamos o símbolo de igual, apenas demos um espaço entre o nome definido e o seu valor:

```
#include <stdio.h>

#define NUMERO_DE_TENTATIVAS 3

int main() {
    // ...

    for(int i = 1; i <= NUMERO_DE_TENTATIVAS; i++) {
        printf("Qual é o seu %do. chute? ", i);
        scanf("%d", &chute);
```

```
        printf("Seu %do. chute foi %d\n", i, chute);

        // ...

    }

    // ...
}
```

Repare que, diferente do que estávamos acostumados com nomes de variáveis, usamos letras em maiúsculo e o símbolo de *underscore* para nomear constantes. Não é obrigatório, mas sim uma convenção. Você verá que convenções como essa são bastante comuns no mundo de desenvolvimento de software. Sempre que cair em uma nova linguagem, descubra as suas convenções: como nomear variáveis, onde colocar o abre e fecha chaves etc.

4.5 O else if

Até agora, usamos `ifs` e `elses` para dizer se o número era maior, menor ou igual do que o número secreto. Veja o código atual. Repare que temos um `if`, um `else` e, dentro desse `else`, um outro conjunto de `if-else`:

```
if(acertou) {
    printf("Parabéns! Você acertou!\n");
    break;
} else {
    int maior = chute > numerosecreto;
    if(maior) {
        printf("Seu chute foi maior do que o número secreto!\n");
    } else {
        printf("Seu chute foi menor do que o número secreto!\n");
    }
}
```

Poderíamos ter escrito tudo isso em uma única combinação de `if-else if-else`. Ou seja, a instrução `else` pode ser combinada com a instrução `if`, justamente para tratar casos como esses, no qual temos várias condições que devem ser avaliadas, até que alguma seja verdadeira.

É esse o caso. Precisamos verificar se os números são iguais (`if`). Se sim, o usuário acertou. Se não, precisamos avaliar se o número é maior (`else if`). Se sim, o usuário chutou um número maior. Caso contrário (`else`), ele chutou um número menor. Em código, veja:

```
int acertou = chute == numerosecreto;
int maior = chute > numerosecreto;

if(acertou) {
    printf("Parabéns! Você acertou!\n");
    break;
} else if(maior) {
    printf("Seu chute foi maior do que o número secreto!\n");
} else {
    printf("Seu chute foi menor do que o número secreto!\n");
}
```

Agora, observe o código a seguir, muito similar ao anterior. Ambos têm o mesmo resultado final, mas seu comportamento é diferente.

```
int acertou = chute == numerosecreto;
int maior = chute > numerosecreto;
int menor = chute < numerosecreto;

if(acertou) {
    printf("Parabéns! Você acertou!\n");
    break;
}
if(maior) {
    printf("Seu chute foi maior do que o número secreto!\n");
}
if(menor) {
    printf("Seu chute foi menor do que o número secreto!\n");
}
```

Quando temos `if-else ifs-else`, assim que a primeira condição for verdadeira, as outras não serão avaliadas. Ele vai de `if` a `if`, e pode eventualmente nem olhar para os que estão mais abaixo. Já no caso dos vários `ifs` em

sequência, mesmo que a primeira dê verdadeiro (suponha que o jogador acertou), os `ifs` debaixo também serão avaliados; ou seja, o computador olhará se o número é maior, e depois se é menor.

Apesar de o comportamento final ser o mesmo (afinal, por lógica, se uma dessas condições forem verdadeira, as outras serão sempre falsas), é importante você entender a diferença entre ambas as abordagens. Com certeza, você passará por momentos nos quais `if-elseifs-else` será a melhor abordagem, e por outros em que `ifs` sequenciais serão a melhor solução.

4.6 Break e continue

O usuário pode digitar o número que quiser. Isso quer dizer que nada o impede de digitar um número negativo, por exemplo. Usuários que digitam valores inválidos são bem comuns em todo sistema de software, e você, programador, deve impedir que eles insiram dados inválidos na aplicação.

Nosso jogo deve mostrar uma mensagem de erro para ele, dizendo que não pode digitar números negativos. Ele também não perderá uma rodada se fizer isso. Fazer isso não é difícil com tudo o que já aprendemos. Um simples `if` verificando se o chute é menor que 0, logo após capturarmos o número do usuário, é suficiente. Repare também que fazemos um `i--`, oposto do `i++`, para subtrair uma unidade da variável, já que essa rodada não vale:

```
for(int i = 1; i <= 3; i++) {
    printf("Qual é o seu %do. chute? ", i);
    scanf("%d", &chute);

    if(chute < 0) {
        printf("Você não pode chutar números negativos\n");
        i--;
    }

    // ...
}
```

Só tem um problema: se o jogador chutar um número negativo, o programa o avisará de que isso é inválido e também dirá que o número é menor

do que o secreto. Não queremos isso. Queremos parar a execução do loop ali, caso o número seja inválido. Mas perceba que essa é uma situação diferente daquela em que usamos o `break`. Ali queríamos matar o loop inteiro. Aqui, queremos apenas passar para a próxima iteração do loop. Para isso, usaremos a instrução `continue`.

O `continue` diz ao laço que ele deve ir direto para a próxima iteração, sem executar o restante de código que ainda possa existir dentro do seu bloco de código. Veja:

```
if(chute < 0) {
    printf("Você não pode chutar números negativos\n");
    i--;
    continue;
}
```

Pronto. Agora, se o usuário inserir dados inválidos, a aplicação vai rejeitá-los.

Portanto, podemos manipular o loop, usando `break` e `continue`. Eles são instruções especialmente úteis para escrevermos códigos mais sucintos. Muitas vezes temos condições a tratar, e elas acabam pode decidir se devemos continuar executando nosso código. Repare que o mesmo código poderia ter sido escrito sem o uso de `continue`. Bastaria fazer um `if-else`. Observe:

```
if(chute < 0) {
    printf("Você não pode chutar números negativos\n");
    i--;
} else {
    printf("Seu %do. chute foi %d\n", i, chute);

    int acertou = chute == numerosecreto;
    int maior = chute > numerosecreto;

    if(acertou) {
        printf("Parabéns! Você acertou!\n");
        break;
    } else if(maior) {
        printf("Seu chute foi maior do que o número secreto!\n");
    } else {
```

```
        printf("Seu chute foi menor do que o número secreto!\n");
    }
}
```

Mas, sempre que colocamos um `if`, precisamos criar um novo bloco de código, uma indentação à frente do resto. Isso pode deixar o código mais difícil de ser lido. Você verá que fugirá cada vez mais de muitas instruções aninhadas (ifs, dentro de ifs, dentro de fors, dentro de ifs etc.). Nosso código com o `continue` é mais sucinto: ele decide se o resto do bloco deve ou não ser executado.

Existem muitas maneiras de escrever o mesmo programa. Isso não é legal?

4.7 O loop while

Vamos mudar a regra do nosso jogo. O jogador não terá mais uma quantidade máxima de jogadas a fazer; elas serão infinitas. Precisamos apenas contá-las, para exibir a ele o número de rodadas que precisou para adivinhar o número. Ainda precisamos de um loop, mas o problema é que agora não sabemos quando ele parará. Em código, não sabemos qual condição colocar dentro do `for`.

O `for` é um ótimo loop quando sabemos a quantidade de vezes que queremos executar alguma coisa. Mas quando não sabemos bem, outros loops caem melhor. Para esse código, o `while` é mais adequado. O `while` também é um loop, com a diferença de que não precisamos saber exatamente a quantidade de vezes que o executaremos.

Queremos fazer algo como:

```
while(JOGADOR NAO ACERTOU) {
    // continua jogando
}
```

Nossa estratégia será a seguinte. Teremos uma variável chamada `acertou`, do lado de fora do `while`, que dirá se o jogador acertou ou não. Obviamente, ela será inicializada com `0`, indicando que ele não acertou. Também teremos uma variável chamada `tentativas`, que guardará o número de chutes feitos pelo usuário (antes, isso estava na variável `i` do `for`, que não existe mais):

```
int numerosecreto = 42;
int chute;
int ganhou = 0;
int tentativas = 1;
```

Em seguida, escrevemos nosso loop. Ele deve acontecer enquanto o jogador não acertou. Já fizemos um `if` semelhante, onde víamos se o jogador acertou, o `if(ganhou)`. Se quiséssemos fazer exatamente o contrário, poderíamos "negar" o `if`, usando o sinal de exclamação, ou seja, `if(!ganhou)`. Nesse caso, ele só entra se o jogador não acertou. Podemos fazer a mesma coisa no `while`. Veja que a exclamação inverte a condição e é fácil de ler: sempre que vir uma exclamação, leia a palavra "não". No loop a seguir, leia "while não ganhou", ou "enquanto não ganhou":

```
int ganhou = 0;
int tentativas = 1;

while(!ganhou) {

}
```

Vamos agora capturar a entrada do usuário e verificar se o número é negativo ou não. Aqui ainda usaremos o `continue`; afinal, ele também funciona com o laço `while`. Repare que trocamos `i` por `tentativas`:

```
int ganhou = 0;
int tentativas = 1;

while(!ganhou) {

    printf("Qual é o seu %do. chute? ", tentativas);
    scanf("%d", &chute);

    if(chute < 0) {
        printf("Você não pode chutar números negativos\n");
        continue;
    }

    printf("Seu %do. chute foi %d\n", tentativas, chute);
```

```
}
```

O bloco de `ifs` e `elseifs` continua igual. Mas repare que retiramos o `break`, e trocamos por um `ganhou = 1`. Afinal, é ele que fará o laço parar:

```
int ganhou = 0;
int tentativas = 1;

while(!ganhou) {

    printf("Qual é o seu %do. chute? ", tentativas);
    scanf("%d", &chute);

    if(chute < 0) {
        printf("Você não pode chutar números negativos\n");
        continue;
    }

    printf("Seu %do. chute foi %d\n", tentativas, chute);

    acertou = chute == numerosecreto;
    int maior = chute > numerosecreto;

    if(acertou) {
        printf("Parabéns! Você acertou!\n");
        ganhou = 1;
    } else if(maior) {
        printf("Seu chute foi maior do que o número secreto!\n");
    } else {
        printf("Seu chute foi menor do que o número secreto!\n");
    }

}
```

Por fim, basta apenas incrementarmos a variável `tentativas`. Antes, a variável `i` era incrementada no terceiro bloco do `for`, executado automaticamente. No `while`, se você precisar contar a quantidade de iterações, precisará fazer isso por conta própria:

```
while(!ganhou) {
    // ...

    if(acertou) {
        printf("Parabéns! Você acertou!\n");
        ganhou = 1;
    } else if(maior) {
        printf("Seu chute foi maior do que o número secreto!\n");
    } else {
        printf("Seu chute foi menor do que o número secreto!\n");
    }

    tentativas++;
}
```

Nosso jogo agora tem tentativas ilimitadas.

Portanto, temos diferentes maneiras de executar o mesmo trecho de código. Chamamos isso de loops. O `for` e `while` são bastante comuns. Usamos `for` quando sabemos a quantidade de vezes que vamos executar o trecho de código, e `while` quando não sabemos bem. Você consegue escrever qualquer loop usando qualquer uma dessas duas abordagens, mas com o tempo, a experiência lhe mostrará qual é o melhor deles, em termos de manutenção. Nesse momento, siga a regra dita anteriormente. Sabe a quantidade? `for`. Não sabe? `while`.

4.8 Loops infinitos

Se você olhar o código com carinho, verá que poderíamos escrever esse mesmo código sem precisar da variável `ganhou`. Afinal, poderíamos fazer uso do `break` e parar o loop no momento em que ele ganhou:

```
while(!ganhou) {
    // ...

    if(acertou) {
        printf("Parabéns! Você acertou!\n");
        break;
    } else if(maior) {
```

```
        printf("Seu chute foi maior do que o número secreto!\n");
    } else {
        printf("Seu chute foi menor do que o número secreto!\n");
    }

    tentativas++;
}
```

Perceba que, com esse código, nosso loop virou **infinito**. Afinal, sua condição nunca será falsa. Nosso programa só para porque existe um `break` ali.

Nesse caso, a variável `ganhou` deixa de ser útil, já que ela será `0` para sempre. Podemos descartá-la e fazer o loop `while` ser infinito, mas sem precisar dela. Sabemos que a condição, quando avaliada, nos devolve `0` ou `1`. O loop itera sempre que a condição é avaliada em `1`. Vamos colocar esse valor direto no loop:

```
while(1) {
    // ...
}
```

Pronto. Agora reescrevemos nosso algoritmo, de maneira a não precisar mais da variável `ganhou`. Veja só como programar é legal: você pode escrever o mesmo programa, de várias maneiras diferentes.

Cuidado com o loop infinito

Um dos grandes perigos quando usamos loops é justamente quando escrevemos loops que não param nunca. Por exemplo, veja o código a seguir, onde usamos um simples `while`:

```
int i = 1;
while(i > 10) {
    printf("i vale %d\n", i);
}
```

Repare que, nesse caso, `i` nunca será maior que 10, pois ele valerá 1 para sempre. Como a condição será sempre falsa, nesses casos, nosso programa ficará infinitamente dentro desse loop.

Isso raramente é proposital. Geralmente, quebramos o loop em algum momento. Portanto, preste atenção na sua condição e garanta que um dia ele parará de executar.

Está perdido?

O código até o momento está em http://pastebin.com/Zhp0XzSc

4.9 Resumindo

Neste capítulo, aprendemos:

- A tomar decisões em nossos programas, usando `ifs`;
- A tomar muitas decisões, usando `ifs`, `else ifs` e `elses`;
- A fazer loops contados com o `for`;
- A fazer loops com `while`;
- A criar constantes usando `#define`.

Capítulo 5

Tipos de dados e operações matemáticas

Como vimos, toda declaração de variável precisa vir acompanhada de um tipo. Não existem variáveis sem tipo. A linguagem C possui vários tipos diferentes, de acordo com o dado que você quer guardar. Você já viu que se quisermos guardar números inteiros, usamos o tipo `int`.

Veja, por exemplo, que em nosso jogo, já declaramos algumas variáveis do tipo inteiro:

```
int numerosecreto = 42;
int chute;
int acertou = 0;
int tentativas = 1;
```

Repare que inicializamos na mesma linha da declaração em várias delas.

Já na variável `chute`, não. Lembre-se sempre de inicializar a variável antes de usá-la; seja na mesma linha, ou algumas linhas depois, usando o sinal de atribuição (=).

5.1 Operações matemáticas

Vamos colocar uma nova regra em nosso jogo: pontos. Ao final da partida, diremos ao jogador quantos pontos ele fez. A regra será a seguinte: todo jogador começa com 1000 pontos. A cada rodada, subtraímos desse número a diferença entre o número secreto e o chute, dividido por 2.

Por exemplo, se o número secreto é 20 e ele chutar 30, subtrairemos 5, que é o resultado de `(30-20)/2`. Se ele chutar 50, subtrairemos 15, e assim por diante. Ou seja, quanto mais perto for o chute dele, menos pontos ele perderá, tendo mais pontos no final.

Para isso, vamos declarar a variável `pontos`, e inicializá-la com o valor 1000:

```
int pontos = 1000;
```

Agora basta manipular essa variável. Como ela guarda um número inteiro, podemos fazer operações matemáticas. Podemos somar (+), multiplicar (*), dividir (/) e subtrair (-) o conteúdo dessas variáveis. Observe:

```
pontos = pontos + 2;
pontos = pontos * 4 / 3;
pontos = pontos - 1;

int outravariavel = pontos * 7;

printf("Você fez %d pontos", pontos);
```

Veja que, nesse código, fizemos diversas operações matemáticas quaisquer e a guardamos na própria variável, ou em uma nova. Estamos programando, logo temos a liberdade de fazer o que quisermos. Você também pode usar parênteses para definir precedência de operações. Obviamente, assim como na matemática, eles são opcionais, e você pode usar quantos quiser:

```
int pontos = ((4 * 2) + (1/2)) * 3 - 1;
```

Agora, basta fazermos a subtração logo antes de o nosso loop acabar. Descobriremos quantos pontos precisamos subtrair e o faremos:

```
for {

    // ...

    int pontosperdidos = (chute - numerosecreto) / 2;
    pontos = pontos - pontosperdidos;

}
```

Vamos também exibir a quantidade de pontos ao final do jogo:

```
for {
    // ...
}

printf("Você fez %d pontos\n", pontos);
printf("Obrigado por jogar!\n");
```

Tudo funciona. Nosso jogo está bem mais divertido!

O que acontece se eu usar uma variável não inicializada?

Depende muito da linguagem e do compilador. Em C, quando você declara uma variável, o programa separa um espaço de memória que estava livre para ela. Se você usá-la sem dar um valor inicial, provavelmente seu conteúdo será um lixo qualquer; o que estava naquela região de memória antes de seu programa pegar para ele, que era usado por algum outro no seu computador.

Linguagens como Java, por exemplo, não permitem que você faça isso. Se você tentar usar uma variável sem inicializá-la, o compilador dará um erro de compilação.

Em C, tudo é mais manual. Portanto, lembre-se de sempre inicializar suas variáveis.

5.2 Outros tipos numéricos

Mas o que acontece se jogarmos os números 45 e 42? O resultado deveria ser 998.5. Afinal, `(45-3)/2` é igual a 1.5. Porém, se rodarmos, nosso jogo nos dará 999 pontos.

O que aconteceu? Simples. Números inteiros não fazem operações com casas decimais, portanto truncam o resultado. Precisamos mudar o tipo da variável `pontos`.

Além de `int`, temos outros tipos para tratar outros problemas. O tipo `double`, por exemplo, nos é útil quando queremos guardar números com casas decimais. Para declarar uma variável desse tipo, reproduziremos o que já fizemos. Veja o código a seguir, que usa tanto doubles quanto inteiros:

```
double pi = 3.1415;
int raio = 3;

double areadocirculo = pi * raio * raio;
printf("A área do círculo é %d", areadocirculo);
```

Outro detalhe importante é o tamanho da variável. Afinal, cada tipo tem

um tamanho definido, e isso implica em guardar números entre certos intervalos. Por exemplo, um inteiro em uma máquina de 32 bits é geralmente armazenado em 4 bytes (ou seja, 32 bits). Isso significa que podemos guardar números de `2^-32` até `2``. São números grandes, mas às vezes não suficientes para o seu tipo de aplicação.

Um bom exemplo é o próprio Youtube, que usava inteiros para guardar o número de vezes que um vídeo foi visto. Em 2014, o clip da música *Gangnam Style*, do coreano *Psy*, ultrapassou o limite dos inteiros; assim, eles precisaram mudar seus tipos para `long`.

O `long` é similar ao inteiro, mas aguenta números maiores ainda; afinal, ele é armazenado em 8 bytes (64 bits). Portanto, ele suporta números de `2^-64` até `2``, sendo estes realmente grandes. Analogamente, temos o `short`, que é um inteiro menor ainda; e o `float`, que é um `double` menor (`float` tem menos precisão do que `double`, ou seja, suporta menos casas decimais).

Todos eles são usados da mesma maneira:

```
int n1;
double n2;
long n3;
short n4;
float n5;
```

Dependendo da plataforma na qual você está desenvolvendo, escolher o tipo mais econômico pode fazer sentido. Muitos desses dispositivos de pequeno porte têm memórias extremamente limitadas.

E para guardar uma palavra?

C não tem um tipo nativo para se guardar uma palavra inteira. Mais à frente, estudaremos arrays de *chars*, usados para resolver tal problema. Nesse momento, não se preocupe com isso.

Em nosso jogo, como a quantidade de pontos pode ter casas decimais, precisamos usar `double`. Assim, poderemos ter números quebrados. Pre-

cisaremos acertar também o `printf`, pois agora precisamos imprimir um `double`. Basta trocarmos o `d` por `f` na máscara:

```
double pontos = 1000;

printf("Você fez %f pontos", pontos);
```

Porém, o estranho é que, se rodarmos novamente nosso jogo, ainda teremos o mesmo problema: 999 pontos. Mas o que aconteceu, se agora definimos a variável como `double`? Observe a linha em que fazemos a operação:

```
double pontosperdidos = (chute - numerosecreto) / 2;
```

Neste momento, você precisa pensar como um compilador. Acredite ou não, ele começa a fazer as operações da direita para esquerda. Ou seja, ele vai primeiro fazer a conta `(chute - numerosecreto) / 2`, para só depois ver que precisa guardar esse resultado na variável `pontosperdidos`.

Olhe só a parte da direita. Veja que `chute`, `numerosecreto` e `2` são inteiros. O que nosso programa fará? Uma conta com números inteiros! A divisão será truncada do mesmo jeito. Veja que esse problema não aconteceria se todas as variáveis ali já fossem `doubles`. Por exemplo, a conta a seguir dá `1.5`, pois todas as variáveis são do tipo certo:

```
double a = 3;
double b = 2;

double c = a / b;
printf("%f", c); // imprime 1.5
```

Precisamos falar para o nosso programa que aquela conta deve ser feita com `double`. Uma possível maneira é convertermos apenas o número `2` para `double`. Se temos uma operação que envolve inteiros e números com pontos flutuantes, ele fará a conta com número flutuante. Logo, basta colocarmos `2.0` em vez de `2`:

```
double pontosperdidos = (chute - numerosecreto) / 2.0;
```

Agora sim, temos números quebrados em nosso jogo.

E SE EU NÃO QUISER IMPRIMIR TANTAS CASAS DECIMAIS?

O C automaticamente imprime várias casas decimais quando usamos a máscara `%f` no `printf()`. Mas podemos mudar isso. Se usarmos a máscara `%.2f`, estamos dizendo que queremos imprimir aquele número com apenas 2 casas decimais:

```
printf("Você fez %.2f pontos\n", pontos);
```

Você pode trocar o número 2 pela quantidade de casas decimais que você quiser.

5.3 Conversões e casting

Na seção anterior, forçamos o número 2 a ser um `double`, ao escrevermos 2.0. Mas será que essa é a única maneira de fazermos isso? E se a nossa expressão fosse composta apenas de variáveis do tipo inteiro? Como forçaríamos a conta para que ela fosse feita com casas decimais?

Veja o código adiante. Nele, teremos o mesmo problema que tivemos com os pontos em nosso jogo. A operação acontecerá primeiro do lado direito e, como todo ele é composto por números inteiros, não haverá casas decimais, mesmo que o tipo da variável à esquerda seja `double`:

```
int a = 3;
int b = 2;

// mesmo problema do nosso jogo
double resultado = a / b;
printf("%f", resultado);
```

Nesse caso, precisamos "converter" as variáveis inteiras para algum tipo com ponto flutuante. A essa conversão, damos o nome de **casting**. Fazê-la é bem fácil: basta colocarmos o tipo para o qual queremos converter entre parênteses, antes de usar a variável. Veja:

```
int a = 3;
int b = 2;
```

```
// o casting acontecendo abaixo nas
// duas variáveis: a, b.
double resultado = (double)a / (double)b;
printf("%f", resultado);
```

Agora sim. Ao colocar `(double)`, estamos dizendo ao compilador para tratar aquela variável, que é um `int`, como `double` naquele momento. A operação acontecerá da forma que queremos.

A operação de casting precisa ser usada com cuidado. Alguns compiladores deixarão você converter de qualquer tipo para qualquer outro tipo. Entretanto, alguns castings podem não ter os resultados esperados.

Por exemplo, se fizermos o casting de um `double` para um `int`, provavelmente perderemos informações. Afinal, um `double` consegue guardar números muito maiores e com casas decimais do que um `int`. Se forçarmos a conversão, a máquina fará, porém ocorrerá uma perda. Veja o código a seguir, no qual convertemos um `double` para `int`. A saída deste programa é 3, pois as casas decimais foram perdidas:

```
// aqui a casa decimal está guardada
double pi = 3.1415;

// agora o casting forçado, de um tipo
// que aguenta mais dados para um que aguenta
// menos dados, perde informação
int coitadodopi = (int)pi;

printf("%d", coitadodopi);
```

Portanto, castings são capazes de converter um tipo em outro. Mas, dependendo do sentido em que isso é feito, informações podem desaparecer. Basta você pensar: se estou convertendo para um tipo em que cabe mais informação do que o tipo atual, ele acontecerá sem problemas; caso contrário, haverá perda de conteúdo.

De volta ao jogo, poderíamos simplesmente alterar a linha da conta dos pontos para alguma das que seguem, e o efeito ainda seria o mesmo:

```
// casting só no 2
double pontosperdidos = (chute - numerosecreto) / (double)2;

// casting nas duas operações
double pontosperdidos =
            (double)(chute - numerosecreto) / (double)2;
```

5.4 Funções matemáticas

Ainda temos um pequeno problema. O que acontece com nossos pontos se chutarmos o número 30? A conta que ele fará é: `(30 - 42) / 2.0`, que dá `-6`. Depois, quando ele for subtrair da variável `pontos`, ele fará `-(-6);` ou seja, somará 6. O jogador ganhará pontos em vez de perder! Precisamos corrigir isso. A primeira solução é fazer um `if`. Se o número for negativo, multiplicamos por `-1` (aí ele ficará positivo). Veja:

```
double pontosperdidos = (chute - numerosecreto) / 2.0;
if(pontosperdidos <0) {
    pontosperdidos = pontosperdidos * -1;
}
```

Esse código funciona, mas temos uma solução mais elegante para isso, em C. Podemos usar a função `abs()`, que recebe um inteiro como parâmetro, e o transforma em positivo. Ou seja, no fim, ela faz a mesma coisa que nosso `if`. A diferença é que gastaremos menos linhas de código para isso.

Veja exemplos:

```
int a = abs(-10); // retorna 10
int b = abs(10);  // retorna 10

int x = -10;
int y = abs(x); // retorna 10
```

Com essa função em mãos, basta agora aplicarmos à operação de subtração, garantindo que ela sempre nos retornará um número positivo. Em código:

```
double pontosperdidos = abs(chute - numerosecreto) / 2.0;
```

A função `abs()`, assim como a `printf`, precisa ser importada de outro arquivo. Fazemos isso por meio de `include`, igual fizemos com o arquivo `stdio.h`. Para usarmos a `abs()`, precisamos importar `stdlib.h`, que significa *standard library*. Basta adicionarmos outra linha de inclusão no começo do arquivo:

```
#include <stdio.h>
#include <stdlib.h>
```

Tanto a *stdlib* quanto a *stdio* são bastante comuns em programas C, já que elas contêm as definições das funções mais comuns. Você verá que seus programas muito provavelmente sempre começarão importando esses dois arquivos.

5.5 Números randômicos

Por enquanto, o número secreto está fixo, e isso tira toda a graça do jogo. Precisamos fazer com que, a cada vez que o executarmos, o computador pense em um número diferente. Sorte a nossa que a *stdlib* contém uma função só para isso: uma que nos devolve um número randômico.

A função chama-se `rand()` e ela nos devolve um número aleatório a cada vez que é executada. Veja:

```
int n1 = rand();
printf("%d", n1);
int n2 = rand();
printf("%d", n2);
```

Ao executar o programa a seguir, ele nos imprime:

```
16807
282475249
```

Dois números diferentes, exatamente como queríamos. Mas o problema acontece se rodarmos de novo: ele nos dará os mesmos números! Ou seja, eles não são tão randômicos assim. Computadores não são capazes de gerar números 100% randômicos. Por isso, dizemos que eles são *pseudorrandômicos*.

Normalmente, as implementações de funções que devolvem esses números usam alguma função matemática bastante complicada para gerar um número que seja difícil de prever.

Para garantirmos que essa função matemática usada pela `rand()` seja diferente a cada rodada, precisamos definir uma "semente" diferente a cada execução do nosso programa. A semente é como se fosse uma variável que entra nessa fórmula matemática e deixa os números sempre diferentes. Mas precisamos de uma semente diferente toda vez, pois se passarmos sempre a mesma, a fórmula também será sempre a mesma, assim como o resultado.

Para isso, utilizamos a data completa atual como semente. Veja que, se rodarmos o programa duas vezes, a data atual sempre será diferente, pois a data exata terá mudado (alguns milissegundos, segundos ou minutos a mais, por exemplo). Isso é suficiente para termos sementes diferentes.

Data é algo bem interessante em sistemas de computação. Uma maneira muito comum de se olhar para ela é pensar nos segundos que se passaram desde uma data específica. Afinal, tendo o número de segundos passados, é possível saber a data exata. Em computação, a data escolhida para tal é "1 de janeiro de 1970"; ela é, inclusive, chamada de *Epoch*. Em C, temos a função `time()`, que nos devolve exatamente essa quantidade de segundos. Para isso, basta declararmos um inteiro e invocarmos a função `time()`, passando o valor "zero" para ela. Em seguida, ela é passada para a função `srand()`, que é quem define a semente.

A função `time()` está definida em outro arquivo, o `time.h`, e também precisamos importá-lo no começo do arquivo. O código completo fica assim:

```
#include <time.h>

int segundos = time(0);
srand(segundos);

int n1 = rand();
printf("%d", n1);
```

Agora, cada vez que executarmos esse programa, teremos um número diferente. Sabendo disso, vamos usar números randômicos em nosso jogo, para que ele fique mais divertido. Porém, ainda temos um detalhe a acertar:

você reparou que esse número randômico é muito grande? Isso não ajudará nosso jogador.

Precisamos definir um intervalo para ele. Ou seja, a máquina só poderá chutar números de 0 a 99. Não conseguimos mudar o comportamento da função `rand()`, mas podemos fazer operações com o resultado. Se calcularmos o resto da divisão desse número por 100, por exemplo, teremos sempre um intervalo entre 0 e 99. Logo, será isso o que faremos. Vamos utilizar o resto da divisão para termos um número no intervalo que queremos. Para calcular o resto da divisão entre dois inteiros, basta usarmos o operador % (porcento):

```
int segundos = time(0);
srand(segundos);

int numerogrande = rand();
int numerosecreto = numerogrande % 100;
```

Ótimo. Nosso jogo está cada vez melhor.

Precisamos de tantas variáveis assim?

Não. Poderíamos ter feito em menos linhas. Por exemplo:

```
srand(time(0));
int numerosecreto = rand() % 100;
```

Isso é questão de gosto. Às vezes, poucas linhas são mais fáceis de serem lidas. Mas, em outras, ter essas variáveis no meio, com nomes bem definidos, ajudam o programador a entender melhor o que está acontecendo.

Comece a pensar nisso. Mostre seu código para seu amigo e veja se ele entendeu. Caso não tenha entendido, ou tenha levado muito tempo para entender, é sinal de que dá para melhorar.

Está perdido?

Você pode ver o código feito até esse momento em
http://pastebin.com/P1mREDbg

5.6 Resumindo

Neste capítulo, aprendemos:

- Que existem tipos de dados diferentes;
- Inteiros armazenam apenas números inteiros, enquanto `doubles` guardam números com ponto flutuante;
- Que podemos fazer as mais diversas operações matemáticas com essas variáveis;
- Usar funções da *stdlib*, como a `abs()`;
- Gerar números randômicos, usando `srand()` e `rand()`;
- Que podemos fazer conversões entre tipos, usando casting;
- Usar resto da divisão para gerar números dentro de um intervalo limitado.

Capítulo 6

Finalizando o jogo

Falta pouco para terminarmos nosso jogo. Vamos mudar agora a regra do número de tentativas. O usuário poderá escolher entre 3 níveis de dificuldade: fácil, médio e difícil. De acordo com o nível, decidiremos a quantidade. Por exemplo, no nível fácil, ele terá 20 chances; no médio, terá 15 chances; e no difícil, apenas 6 chances.

Com o que vimos até aqui, basta perguntarmos para o usuário o nível de dificuldade, aí usamos alguns `if`s para decidir a quantidade:

```
int nivel;
int totaldetentativas;

printf("Qual o nível de dificuldade?\n");
printf("(1) Fácil (2) Médio (3) Difícil\n\n");
printf("Escolha: ");
```

```
scanf("%d", &nivel);

if(nivel == 1) {
    totaldetentativas = 20;
} else if (nivel == 2) {
    totaldetentativas = 15;
} else {
    totaldetentativas = 6;
}
```

Como sabemos a quantidade de vezes que o loop tem de acontecer, basta trocarmos o `while` atual por um `for`. Precisamos também lembrar de jogar fora a variável `tentativas` e trocar todas as vezes que a usamos por `i`. Afinal, `i` voltou a ser nosso contador da jogada. Precisamos também voltar com o `break` para caso o usuário acerte o número, pois nosso `for` agora só parará quando o número de tentativas esgotar. Por fim, precisamos voltar o `i--` caso ele chute um número negativo:

```
// deletamos a declaração do acertou aqui de cima

for(int i = 1; i <= totaldetentativas; i++) {

    printf("Tentativa %d de %d\n", i, totaldetentativas);

    // ...

    if(chute < 0) {
        printf("Você não pode chutar números negativos\n");
        i--;
        continue;
    }

    // ...

    acertou = chute == numerosecreto;
    if(acertou) {
        printf("Parabéns! Você acertou!\n");
        break;
    }
```

```
    // ...
}
```

> **Está perdido?**
>
> Fizemos bastante mudanças nesse código. Variáveis sumiram, outras apareceram. É um bom desafio para você fazer o jogo compilar e funcionar até aqui.
>
> Se está se sentindo perdido, veja o código feito até esse momento em http://pastebin.com/yzaRVAhX

6.1 Switch e case

Uma alternativa aos `ifs` é usar o `switch`. O `switch` é um *if mais simples*. Quando temos condições repetidas, nas quais comparamos uma variável inteira com vários números (igual em nosso caso, em que comparamos `nivel` com 1, 2 e 3), podemos usar o `switch`.

Sua sintaxe é mais clara para esse tipo de decisão. Primeiro, você define em qual variável o `switch` acontecerá. Em seguida, você tem uma sequência de `case`, um para possível opção. Após os dois-pontos, você escreve o bloco de código que será executado, caso o `case` seja verdadeiro. Um detalhe importante é que, ao final de cada bloco, você não pode se esquecer do `break` (sim, o mesmo que usamos no loop), pois precisamos quebrar a execução do `switch`. Veja:

```
int nivel;
scanf("%d", &nivel);

switch(nivel) {
    case 1:
        totaldetentativas = 20;
        break;
    case 2:
        totaldetentativas = 15;
        break;
```

```
    case 3:
        totaldetentativas = 6;
        break;
}
```

Esse `switch` trata os casos de números 1, 2 e 3. Se o usuário digitar, por exemplo, 4, o `switch` não fará nada. Podemos definir o comportamento padrão; ou seja, o que ele deve fazer, caso nenhum `case` bata. Vamos definir o seguinte comportamento: se o usuário digitar qualquer outro número que não 1, 2, 3, o nível será avançado. Isso quer dizer que precisamos transformar o `case 3` em um `else`. O `else` do `switch` é o `default`. Observe:

```
int nivel;
scanf("%d", &nivel);

switch(nivel) {
    case 1:
        totaldetentativas = 20;
        break;
    case 2:
        totaldetentativas = 15;
        break;
    default:
        totaldetentativas = 6;
        break;
}
```

6.2 Novamente usando variáveis e escopos

Nosso jogo ainda tem um *bug*. Você conhece o termo *bug*, certo? É o nome do dia a dia que damos para problemas em nossa aplicação. Se o usuário acerta o número, sempre mostramos uma mensagem amigável a ele. Mas falta mostrarmos uma mensagem avisando que ele perdeu, caso suas tentativas se esgotem.

Nesse momento, se elas se esgotarem, ainda exibimos "Obrigado por jogar"; porém precisamos exibir algo mais claro para ele. Algo como "Você perdeu, tente novamente!". A questão é: onde colocar essa lógica?

Exibir a mensagem de vitória era fácil, pois tínhamos um `if` especial pra esse caso. Mas, no caso da derrota, não. Precisamos esperar o `for` acabar e, aí sim, mostrar a mensagem. O primeiro passo é colocá-la logo após o fim do loop:

```
} // fim do for

printf("Você perdeu! Tente novamente!");

printf("Você fez %.2f pontos\n", pontos);
printf("Obrigado por jogar!\n");
```

Entretanto, desse jeito, essa mensagem aparecerá tanto no caso da vitória quanto da derrota. Precisamos de um `if` ali, que faça com que isso seja impresso apenas na derrota. Podemos usar a variável `acertou`. Veja que, como ela está declarada fora do `for`, logo no começo do nosso programa, podemos usá-la mesmo após o loop. Agora pense, se o loop for até a última iteração, o conteúdo da variável `acertou` será o resultado calculado na última iteração. Se ele venceu o jogo, `acertou` valerá 1; caso contrário, 0. Podemos fazer o loop:

```
int acertou = 0;

for(...) {
    // código aqui

    acertou = chute == numerosecreto;
    int maior = chute > numerosecreto;

    if(acertou) {
        printf("Parabéns! Você acertou!\n");
        break;
    } else if(maior) {
        printf("\nSeu chute foi maior do que o número
                secreto!\n\n");
    } else {
        printf("\nSeu chute foi menor do que o número
                secreto!\n\n");
```

```
    }

    // código aqui
}

if(!acertou) {
    printf("Você perdeu! Tente novamente!\n");
}
```

Ótimo. Agora mostramos uma mensagem amigável na derrota. Já que tudo está funcionando, podemos melhorar um pouco nosso código. Veja que a mensagem de derrota está embaixo do `for`, mas a de sucesso está dentro. Podemos colocar tudo em um só lugar para facilitar a manutenção. Ou seja, saberemos que, logo após o `for`, entra a inteligência da mensagem de ganhou ou perdeu.

Vamos remover o `printf()` que existe dentro do `if(acertou)`, que está dentro do `for`, e colocá-lo em um `else`, logo após o `for`. Repare que, como agora temos `if-else`, a condição foi invertida (fizemos `if(acertou)` em vez de `if(!acertou)`. Você poderia programar de ambos os jeitos, afinal o resultado seria o mesmo. Porém, com certeza é mais fácil entender uma condição sem a negação do que com ela:

```
for(...) {
    if(acertou) {
        break;
    }

    // ...
}

if(acertou) {
    printf("Parabéns! Você acertou!\n");
} else {
    printf("Você perdeu! Tente novamente!\n");
}
```

6.3 Embelezando o jogo

Nosso jogo está inteiro funcional. Mas ainda não está bonito. As frases estão coladas umas às outras, os textos estão repetidos etc. Interface é fundamental para nossos usuários. Afinal, eles não querem saber o que está por dentro; eles só querem interagir com a parte de fora.

Mesmo que estejamos fazendo aplicações no terminal, ainda assim podemos caprichar e fazer nossos programas terem saídas elegantes. O cabeçalho pode ser algo mais chamativo. Podemos fazer desenhos usando só letras! Por exemplo, um cabeçalho mais divertido seria:

```
printf("\n\n");
printf("          P  /_\  P                              \n");
printf("         /_\_|_|_/_\                             \n");
printf("     n_n | ||. .|| | n_n         Bem-vindo ao    \n");
printf("     |_|_|nnnn nnnn|_|_|     Jogo de Adivinhação!\n");
printf("    |" "  |  |_|  |"  " |                        \n");
printf("    |_____| ' _ ' |_____|                        \n");
printf("          \__|_|__/                              \n");
printf("\n\n");
```

> **ASCII Art**
>
> Você consegue facilmente achar desenhos como esse no Google. Basta procurar por "ascii art".

O cabeçalho é muito mais bonito; entretanto, se o compilarmos, teremos problemas. O compilador nos dará um erro parecido com esse:

```
adivinhacao.c:9:25: warning: unknown escape sequence '\ '
   printf("          P  /_\  P                              \n");
                          ^~
```

Ele reclamou do `\`. Este caractere dentro do `printf` é especial. Afinal, quando queremos pular uma linha, usamos `\n`. Mas e se quisermos realmente imprimi-lo? Precisaremos dobrá-lo, ou seja, escrever `\\`. Vamos

corrigir o desenho do nosso castelo, dobrando todas as barras necessárias. A mesma coisa com as aspas, pois precisamos fazer `\"` se quisermos imprimi-las na tela (caso contrário, o compilador pode confundir e achar que aquela aspas está fechando a aspas aberta anteriormente).

Apesar de o desenho ter ficado torto no código, quando o executarmos, ele ficará certo:

```
printf("\n\n");
printf("          P  /_\\  P                              \n");
printf("         /_\\_|_|_/_\\                             \n");
printf("     n_n | ||. .|| | n_n         Bem-vindo ao     \n");
printf("     |_|_|nnnn nnnn|_|_|     Jogo de Adivinhação! \n");
printf("    |\" \"  |  |_|  |\"  \" |                     \n");
printf("    |_____| ' _ ' |_____|                         \n");
printf("          \\__|_|__/                              \n");
printf("\n\n");
```

E agora, já que mostramos a tentativa em que ele está, não precisamos mais ficar mostrando "Qual o seu Xo. chute?". A pergunta pode ser mais simples. Também não é mais necessário exibir o número chutado, afinal o usuário que o digitou, e ele já fica na tela:

```
// antes
printf("Tentativa %d de %d\n", i, totaldetentativas);
printf("Qual é o seu %do. chute? ", i);

// depois
printf("-> Tentativa %d de %d\n", i, totaldetentativas);
printf("Chute um número: ");
```

Vamos também exibir uma mensagem de vitória ou derrota mais bonita, usando ASCII Art novamente. Uma rápida googlada, e temos desenhos de uma cara de felicidade e de uma careta, ideais para a mensagem que queremos passar. Também exibiremos a quantidade de pontos somente se o jogador ganhou o jogo:

```
printf("\n");
if(acertou) {
```

```
    printf("             00000000000               \n");
    printf("         0000000000000000000           \n");
    printf("      000000  000000000  000000        \n");
    printf("    000000      00000      000000      \n");
    printf("  00000000  #   00000  #   00000000    \n");
    printf(" 0000000000    0000000    0000000000   \n");
    printf("0000000000000000000000000000000000000  \n");
    printf("0000000000000000000000000000000000000  \n");
    printf("0000  0000000000000000000000000  0000  \n");
    printf(" 0000  00000000000000000000000  0000   \n");
    printf("  0000   00000000000000000000  0000    \n");
    printf("    00000   000000000000000   0000     \n");
    printf("      000000   000000000   000000      \n");
    printf("         000000         000000         \n");
    printf("             000000000000              \n");
    printf("\nParabéns! Você acertou!\n");
    printf("Você fez %.2f pontos. Até a próxima!\n\n", pontos);
} else {
    printf("       \\|/ ____ \\|/    \n");
    printf("        @~/ ,. \\~@      \n");
    printf("       /_( \\__/ )_\\    \n");
    printf("          \\__U_/         \n");
    printf("\nVocê perdeu! Tente novamente!\n\n");
}
```

6.4 Acabamos o primeiro jogo!

Terminamos nosso primeiro jogo! Viu só como não foi difícil? Mas ainda temos muita coisa para aprender sobre a linguagem C. No próximo capítulo, começaremos o nosso próximo jogo, o jogo de forca. Ele nos fará aprender mais coisas ainda.

Até lá!

Código final

O código final do jogo pode ser visto aqui: http://pastebin.com/SBe41fgB

6.5 Resumindo

Neste capítulo, aprendemos:

- `Switch` e `case` como alternativas para alguns casos do `if`;
- Que precisamos escapar barras e aspas se quisermos imprimi-las no `printf`;
- Que interface é importante e que conseguimos fazer programas bonitos mesmo no console.

Capítulo 7

Exercícios

Agora é sua vez de praticar! Veja os exercícios a seguir e tente resolvê-los. Alguns deles são melhorias no próprio jogo de adivinhação; outros são novos programas, e você deve fazê-los em um arquivo separado.

7.1 Melhorando o jogo de adivinhação

1) Hoje o jogo escolhe um número entre 0 e 99. Deixe o usuário escolher esse limite. Você precisa capturar esses dois números dele, e usá-los na hora de calcular o número randômico.

2) Temos ainda alguns números mágicos em nosso código. Use `#define`s para a quantidade de tentativas por níveis e quantidade de pontos inicial.

3) Ao terminar uma partida, pergunte para o usuário se ele quer jogar novamente. Se ele digitar "1", significa que quer, então você deverá começar o

jogo de novo. Para isso, você precisará usar mais um loop.

4) Não deixe o usuário jogar o mesmo número na sequência. Se ele jogou o número "2", errou, e jogou novamente o "2", avise-o de que ele já jogou esse número, e não conte como uma tentativa. Para isso, crie uma variável que guardará o "último número chutado". *(Não tente guardar todos os números chutados nesse momento. Aprenderemos como fazer isso mais para frente.)*

7.2 Outros desafios

1) Escreva um programa que imprima todos os números pares entre 2 e 50. Para saber se o número é par, basta você ver se o resto da divisão do número por 2 é igual a 0.

2) Escreva um programa que imprima a soma de todos os números de 1 até 100. Ou seja, ele calculará o resultado de `1 + 2 + 3 + 4 + ... + 100`.

3) Escreva um programa que peça um inteiro ao usuário, e com esse inteiro, ele imprima, linha a linha, a tabuada daquele número até o 10. Por exemplo, se ele escolher o número 2, o programa imprimirá: `2x1=2, 2x2=4, 2x3=6, ..., 2x10=20`.

4) Implemente uma calculadora. O programa deve pedir 3 números ao usuário: `a`, `b` e `operacao`. Se `operacao` for igual a 1, você deverá imprimir a soma de `a + b`. Se ela for 2, a subtração. Se for 3, a multiplicação. Se for 4, a divisão.

5) Escreva um programa que peça um número inteiro ao usuário e imprima o fatorial desse número. Para calcular o fatorial, basta ir multiplicando pelos números anteriores até 1. Por exemplo, o fatorial de 4 é `4 * 3 * 2 * 1`, que é igual a 24.

6) Dados três números, imprimi-los em ordem crescente.

Capítulo 8

Jogo de forca

Nosso próximo objetivo é desenvolver um jogo de forca. O computador escolherá, de maneira randômica, uma palavra que está salva em um arquivo

de palavras; e o jogador deve chutar, letra por letra, até adivinhá-la. Se o jogador chutar 5 letras erradas, ele perde. Ao final, o computador oferecerá a possibilidade de o usuário inserir uma nova palavra no banco de dados.

Nestes capítulos, a ideia é ensinar você a:

- Criar, varrer e manipular arrays de diferentes tipos.
- Usar arrays de *chars* (strings) e funções para manipulá-los.
- Entender e criar funções que recebem parâmetros e devolvem valores.
- Ler e escrever arquivos com formatos específicos.
- Aprender mais boas práticas de código, como a criação e extração de funções para evitar repetição de código.
- Criar *header files*.

8.1 Como ele ficará?

Nosso jogo também terá uma interface amigável. Ao iniciar, ele nos dará boas-vindas e imprimirá a força, por enquanto, sem ninguém:

```
/****************/
/ Jogo de Forca */
/****************/


  _______
 |/      |
 |
 |
 |
 |
 |
_|___


_ _ _ _ _ _ _ _ _
Qual letra?
```

Se você chutar uma letra errada, ele o avisará e o homenzinho começará a aparecer na forca:

```
Você errou: a palavra NÃO tem a letra X
```

```
_ _ _ _ _ _ _ _ _
Qual letra?
```

Se você chutar uma letra certa, a letra aparece entre o vazado:

```
Qual letra? A
Você acertou: a palavra tem a letra A
```

```
_ _ _ _ _ A _ _ A
Qual letra?
```

Se você perder, ele te enforcará por completo:

```
 -------
 |/      |
```

Uma caveira também aparecerá, avisando que o jogo acabou:

```
Puxa, você foi enforcado!
```

E se você ganhar, um troféu aparecerá:

```
Parabéns, você ganhou!


       ___________
      '._==_==_=_.'
      .-\:      /-.
     | (|:.     |) |
      '-|:.     |-'
        \::.    /
         '::. .'
           ) (
```

```
  _.'   '._
 '_______'
```

8.2 É hora de começar!

Está pronto para desenvolver esse jogo? **Mãos à obra!**

Capítulo 9

Arrays

Para se começar um jogo de forca, uma maneira análoga ao nosso jogo de adivinhação é guardar a palavra secreta. Até agora, já conhecemos diversos tipos de variáveis, como `int`, `double`, `float`, `long`; porém ainda não vimos o tipo usado para se guardar um texto.

Em C, para guardarmos uma única letra, usamos o tipo `char`. Ele é declarado da mesma forma que os outros. Mas, para se passar a letra a ser armazenada, usamos aspas simples; e para imprimir, usamos a máscara `c` no `printf`. Veja o código a seguir:

```
#include <stdio.h>

int main() {
    char letra1 = 'M';
    char letra2 = 'F';
    char letra3 = 'A';
```

```
    printf("%c %c %c", letra1, letra2, letra3);
}
```

Ele nos imprime as letras “M”, “F” e “A”. Mas e para guardar uma palavra? Será que precisamos armazenar letra por letra? A resposta é: sim. Precisamos guardar uma a uma; porém, para isso, não teremos uma variável diferente para cada letra. Aliás, se tivéssemos, ficaríamos malucos. Imagine para guardar um texto inteiro? Quantas variáveis teríamos?

E o pior, não conseguiríamos fazer códigos dinâmicos para imprimir palavras de tamanhos diferentes. Não podemos escrever um `for`, por exemplo, para imprimir as várias letras:

```
// imagine uma palavra com 10 letras
for(int i = 1; i <= 10; i++) {
    // não temos uma maneira de "gerar dinamicamente"
    // o nome da variável a ser impressa.
    // Ou seja, não conseguimos imprimir "letra1",
    // depois "letra2", e assim por diante.
    printf("%c", letra(i));
}
```

O que usaremos para isso é o que chamamos de **array**. Arrays são uma maneira de guardarmos “vários” de um só tipo. Esses elementos ficam armazenadas na memória, um ao lado do outro. Por exemplo, um array de inteiros guarda um monte de números inteiros. Um array de *char* guarda um monte de caracteres, e assim por diante. A vantagem do array é que, no momento da declaração, dizemos o seu tamanho e acessamos cada um desses elementos por meio de um índice.

Veja, por exemplo, uma declaração de um array de inteiros, com 5 posições. Note que a declaração é idêntica, com exceção do uso dos colchetes:

```
int notas[5];
```

Agora, para usar esse array, guardando e lendo seus inteiros, basta usarmos os colchetes para definirmos a posição que queremos acessar. Afinal, temos 5. O único detalhe importante é que, se nosso array tem 5 posições,

elas vão de 0 a 4. A primeira posição de um array é o 0 (e não o 1, como muitos confundem):

```
// guardando na primeira posição do array
notas[0] = 10;

// guardando na segunda posição
notas[1] = 5;

// imprimindo a primeira posição
printf("%d", notas[0]);
```

9.1 Strings e array de chars

Para guardarmos nossa palavra secreta, vamos declarar um array de char. Vamos definir seu tamanho como 20:

```
char palavrasecreta[20];
```

Podemos guardar nossa palavra dentro dele, da mesma forma como estávamos fazendo antes. A diferença é que o número que passamos entre colchetes pode vir de uma variável:

```
char palavrasecreta[20];

palavrasecreta[0] = 'M';
palavrasecreta[1] = 'E';
palavrasecreta[2] = 'L';

int numero = 3;
palavrasecreta[numero] = 'A';

palavrasecreta[4] = 'N';
palavrasecreta[5] = 'C';
palavrasecreta[6] = 'I';
palavrasecreta[7] = 'A';
```

Um trabalhão. Para imprimir a palavra, fica um pouco mais fácil. Afinal, como podemos usar variáveis ali no índice, conseguimos fazer um loop com o tamanho da palavra:

```
for(int i = 0; i < 8; i++) {
    printf("%c", palavrasecreta[i]);
}
```

Entretanto, um array de char é um array especial. Como nós o usamos para guardar textos, temos funções especializadas para facilitar nossa vida na hora de colocar dados dentro dele. Essas funções também vão nos ajudar a escrever loops mais flexíveis (o 8 ali no código está fixo, mas queremos que ele se adapte ao tamanho da palavra).

A função `sprintf()` é muito parecida com a `printf()`. Enquanto o `printf` imprime na tela, a `sprintf` imprime em um array de chars (o *s* é justamente de *string*). Logo, se quisermos guardar a palavra secreta no array, fazemos:

```
sprintf(palavrasecreta, "MELANCIA");
```

Para imprimirmos na tela, usamos o `printf` com a máscara `s`. Ela faz com que todo o array seja impresso:

```
printf("%s", palavrasecreta);
```

Entretanto, veja que interessante. Nosso array tem 20 posições e a palavra "melancia" tem apenas 8 caracteres. Como que ele soube que deveria parar no último "a"? Por que ele não imprimiu 20 caracteres? Strings em C têm um pequeno detalhe: sempre que guardamos uma palavra lá como melancia, que ocupa 8 posições , guardamos um caractere especial na 9ª posição. Esse caractere nos diz que a string acaba ali, e que o resto é "lixo". Ele é representado pelo símbolo ` (barra zero). Apesar de serem dois caracteres, lembre-se de que o barra é especial. Esse ` é armazenado em uma única posição.

Ou seja, quando fizemos o `sprintf()`, ele escreveu na verdade o texto `MELANCIA``. A função `printf()`, sabendo disso, foi lendo o array de chars até encontrar o `. Encontrou, parou.

Se estivéssemos declarando as letras dessa palavra uma a uma, igual fizemos anteriormente, bastaria adicionar o ` ao final, que o `printf("%s")` funcionaria normalmente:

```
char palavrasecreta[20];

palavrasecreta[0] = 'M';
palavrasecreta[1] = 'E';
palavrasecreta[2] = 'L';
palavrasecreta[3] = 'A';
palavrasecreta[4] = 'N';
palavrasecreta[5] = 'C';
palavrasecreta[6] = 'I';
palavrasecreta[7] = 'A';

// colocando o \0 para indicar que a string acabou
palavrasecreta[8] = '\0';
```

9.2 Varrendo o array

Vamos começar nosso jogo. Assim como o anterior, ele terá um loop principal. A ideia é que, enquanto ele não acertar a palavra, ou não morrer enforcado, continuamos a pedir uma letra a ele. Vamos criar duas variáveis, uma chamada `acertou`, e outra chamada `enforcou`. Ambas serão booleanos; ou seja, guardarão 0 ou 1, indicando se o jogador ganhou ou foi enforcado, respectivamente.

Até então, aprendemos dois tipos de loops: o `for` e o `while`. Ambos, antes de executarem o corpo de código que possuem, eles validam a condição. Isso quer dizer que, se a condição do `while` for falsa desde o começo, o código não será executado nenhuma vez. Mas nesse nosso caso (e no jogo anterior também), ao menos uma vez precisamos rodar o bloco de código.

Podemos fazer uso do `do-while`, um outro tipo de loop, idêntico ao `while`, com a exceção de que ele executa o bloco de código ao menos uma vez. Repare na condição `while(!acertou && !enforcou)`. Conseguimos lê-la em português: *enquanto não acertou E não enforcou*.

```
int acertou = 0;
int enforcou = 0;

do {
```

```
    // código aqui
} while(!acertou && !enforcou);
```

Invertendo booleanos

Perceba que a exclamação pode ser usada para “negar” (ou inverter) o valor de um booleano. Veja o código a seguir, onde declaramos dois inteiros, `verdadeiro` e `falso`, colocando 1 e 0 dentro deles. Em seguida, imprimimos ambos duas vezes; uma vez normal, e a outra usando a exclamação para inverter o valor.

```
int verdadeiro = 1;
int falso = 0;

printf("%d %d", verdadeiro, !verdadeiro);
printf("%d %d", falso, !falso);
```

Repare que o programa imprimirá `1 0` na primeira linha, e `0 1` na segunda. Ou seja, ele inverte o valor. Usamos a exclamação para conseguirmos escrever mais clara e sucintamente expressões booleanas.

O mesmo loop poderia ter sido feito usando a comparação com 0 (falso). Porém, veja como escrevemos mais código dessa forma:

```
while(acertou == 0 && enforcou == 0);
```

Vamos agora capturar uma letra do usuário, e ver se ela existe na palavra secreta. Para isso, precisaremos fazer um `for` na palavra (afinal, sabemos o tamanho dela), e comparar se o `char` que está naquela posição bate com o informado pelo usuário. Para capturar o tamanho da palavra dentro de um array de chars, usamos a função `strlen()`. Ela está disponível em `<string.h>`.

A usaremos no `for`, que irá de 0 (primeira posição do array) até `strlen()`, que é a última posição preenchida. Em seguida, compararemos o conteúdo daquela posição com o chute dado, e avisaremos o usuário se ele acertou o chute:

```
do {
    char chute;

    printf("Qual letra? ");
    scanf("%c", &chute);

    for(int i = 0; i < strlen(palavrasecreta); i++) {
        if(palavrasecreta[i] == chute) {
            printf("A posição %d tem essa letra\n", i+1);
        }
    }
} while (!acertou && !enforcou);
```

A saída do programa, nesse momento, é a seguinte:

```
Qual letra? A
A posição 4 tem essa letra
A posição 8 tem essa letra
Qual letra? Qual letra? M
A posição 1 tem essa letra
Qual letra? Qual letra? E
A posição 2 tem essa letra
Qual letra? Qual letra? L
A posição 3 tem essa letra
```

Ainda longe do ideal, mas já é um começo. Agora, repare com atenção, e veja que ele escreveu "Qual a letra?" duas vezes, várias vezes? Temos um bug aí. Isso aconteceu por um motivo bastante diferente. Veja que sempre que usávamos o `scanf()`, digitávamos um número, por exemplo, e dávamos `enter`. O `scanf` é inteligente e sabe que `enter` não faz parte do número. Mas, agora que estamos lendo um caractere único, um `enter` é considerado um caractere. O `scanf()` se perde quando digitamos uma letra e um `enter`. O `enter` fica de *buffer*, e quando executamos o `scanf()` pela segunda vez, ele nem nos pede a letra e passa o `enter` direto.

Para resolver isso, precisamos mudar um pouco a máscara. Em vez de passarmos apenas o `"%c"`, passaremos `" %c"`. Repare no espaço. Ele diz ao `scanf()` para ignorar o `enter`. Isso resolve o problema:

```
scanf(" %c", &chute);
```

A saída agora, como esperado:

```
Qual letra? A
A posição 4 tem essa letra
A posição 8 tem essa letra
Qual letra? M
A posição 1 tem essa letra
Qual letra? E
A posição 2 tem essa letra
Qual letra? L
A posição 3 tem essa letra
```

Está perdido?

O código até este ponto está em http://pastebin.com/jyLuQWd2

9.3 Laços encadeados

Precisamos agora mostrar para o usuário a palavra, mas de maneira escondida. Ou seja, uma sequência de _ _ _ _, um *underscore* por letra. À medida que o jogador acerta uma letra, abrimos essa casa. Vamos começar devagar, imprimindo a palavra escondida.

O código é parecido com o loop anterior. Precisamos passar por cada letra da palavra secreta, e imprimir o underscore:

```
do {

    for(int i = 0; i < strlen(palavrasecreta); i++) {
        printf("_ ");
    }
    printf("\n");

}
```

Agora, precisamos pensar em abrir a letra que ele já acertou. Para isso, precisamos guardar todos os seus chutes. Para guardar muitos chars, você já

sabe: array. Vamos declarar um array responsável por guardar cada um dos chutes do usuário, e uma variável que conta o número de chutes já dados. Essa variável será especialmente importante, pois, além de indicar a quantidade, ela nos indicará a posição desse array que o último chute dado deve ser guardado.

Por exemplo, imagine o array `chutes` e o inteiro `tentativas`, inicializado com 0. Quando `tentativas` for igual a 0, quer dizer que o usuário deu 0 chutes, e o primeiro chute deve ser guardado em `chutes[0]`. Quando `tentativas` for igual a 1, isso quer dizer que o usuário deu 1 chute, e o próximo chute deve ser guardado em `chutes[1]`. Quando quisermos ver todos os chutes, podemos também fazer um `for` que vai de 0 a `tentativas`.

Vamos começar declarando essas variáveis, antes do `do-while`:

```
char chutes[26];
int tentativas = 0;
```

Agora, vamos começar a salvar os chutes do usuário nesse array. Logo após capturarmos o char no `scanf()`, vamos guardá-lo no array. Já sabemos que a posição livre do array está na variável `tentativas`:

```
char chute;
printf("Qual letra? ");
scanf(" %c", &chute);

chutes[tentativas] = chute;
tentativas++;
```

Agora vamos voltar ao nosso loop anterior que exibe a palavra escondida. Precisamos de alguma inteligência ali: se o usuário já chutou aquela letra, precisamos exibi-la; e se nunca chutou, exibimos o underscore. Para isso, para cada letra da palavra secreta, precisamos fazer um loop dentro do array de chutes, e ver se ela está lá.

Esse loop precisa ter uma variável de fora, tomando conta do resultado. Afinal, se acharmos a letra, em qualquer posição do array, devemos pará-lo e imprimi-la. Caso, após passarmos em todo o array e o loop acabar, e a letra não estiver lá, exibimos o underscore.

Veja, em código, a variável achou, que valerá 1, se acharmos a letra no array de chutes. Vamos colocar a impressão logo no começo do do-while, e remover o for que tínhamos mais embaixo (afinal, precisamos mostrar a forca antes de pedir a próxima letra). Repare também que usamos a letra j para o loop de dentro, visto que a variável i já está sendo usada pelo loop de fora:

```
do {
    for(int i = 0; i < strlen(palavrasecreta); i++) {

        int achou = 0;

        for(int j = 0; j < tentativas; j++) {
            if(chutes[j] == palavrasecreta[i]) {
                achou = 1;
                break;
            }
        }

        if(achou) {
            printf("%c ", palavrasecreta[i]);
        } else {
            printf("_ ");
        }

    }
    printf("\n");
    // ...
}
```

Veja que o for aninhado ou seja, o for que está dentro do for inicializa uma variável j que varia de 0 até a quantidade de tentativas já dadas. Em seguida, ele compara se a letra que está em chutes[j] é igual a que está em palavrasecreta[i]. Repare que j vai variar de 0 até o número de chutes, enquanto i vai valer 0. Depois, j variará novamente de 0 até o número de chutes, enquanto i valerá 1, e assim por diante. Ou seja, compararemos todos com todos.

Vamos fazer o que chamamos de **teste de mesa**, isto é, simular nosso có-

digo no papel. Imagine que a palavra secreta seja `"MAO"`, e que os chutes dados foram “X” e “A” (a quantidade de tentativas, nesse momento, é 2).

1) i = 0, j = 0, palavrasecreta[0] é ‘M’ e chutes[0] = ‘X’. M não é X, o loop continua.

2) i = 0, j = 1, palavrasecreta[0] é ‘M’ e chutes[1] = ‘A’. M não é A, o loop continua.

3) `i` é incrementado.

4) i = 1, j = 0, palavrasecreta[1] é ‘A’ e chutes[0] = ‘X’. A não é X, o loop continua.

5) i = 1, j = 1, palavrasecreta[1] é ‘A’ e chutes[1] = ‘A’. **A é A**, o loop é quebrado, achou = 1, letra é impressa.

6) `i` é incrementado.

7) i = 2, j = 0, palavrasecreta[1] é ‘O’ e chutes[0] = ‘X’. O não é X, o loop continua.

8) i = 2, j = 1, palavrasecreta[1] é ‘O’ e chutes[1] = ‘A’. O não é A, o loop continua.

9) Loop maior termina.

Loops dentro de loops podem parecer complicados no começo. Use e abuse de testes de mesa. Coloque as variáveis que estão definidas naquele momento, e vá trocando uma a uma. Se quiser, coloque `printf`s no código, imprimindo as variáveis `i`, `j`, e quaisquer outras que quiser para ajudá-lo no entendimento.

Neste momento, nosso jogo já se parece mais com um jogo de forca. O usuário consegue chutar letras, e vamos as mostrando conforme ele acerta. Estamos caminhando bem.

Está perdido?

Você pode ver o código até este momento em
http://pastebin.com/ctVzUyqy

9.4 Resumindo

Neste capítulo, você aprendeu:

- O que são arrays;
- Como declarar arrays;
- Como escrever dentro de arrays;
- Como pegar o conteúdo de uma posição do array;
- Que as posições do array começam no 0, e não no 1;
- Que arrays de char são arrays como qualquer outro;
- Que se guardamos uma string em um array de char, o caractere ` deve ser colocado ao fim da palavra;
- Que podemos colocar loops dentro de loops.

Capítulo 10

Números binários

Vimos até então que todas as variáveis que declaramos, sejam elas simples ou arrays, ficam armazenadas na memória do computador. A pergunta é: como ele faz isso? Como ele guarda, por exemplo, um número, dentro daquela peça de hardware que chamamos de memória?

Acredite ou não, uma memória de computador consegue apenas diferenciar duas coisas: se algo está imantado para um lado ou para o outro. A um dos lados atribuímos o valor 1, e ao outro, o valor 0. Ou seja, a memória de um computador não consegue guardar um número como "42", pois ela não consegue guardar nem "4" nem "2". Ela consegue guardar apenas "1"s e "0"s. E aí vem o desafio: precisamos representar qualquer coisa usando apenas esses dois números.

Vamos ver como isso funciona devagar. Imagine o número 0, como representá-lo? Fácil: 0. E o número 1? Também é fácil, 1. Pronto, temos um **bit**.

O desafio começa quando queremos representar o número 2, usando somente os números 0 e 1. A solução é usar o 10 para representar o número 2. Cuidado. Não leia `10` como dez, leia como "um zero", afinal ele não vale "dez". E o número três? 11. Isto é, "um um". E depois o número quatro? 100. Depois 101, 110, 111.

Ficamos com uma tabela:

```
0  =>     0
1  =>     1
2  =>    10
3  =>    11
4  =>   100
5  =>   101
6  =>   110
7  =>   111
8  =>  1000
...
```

Repare que a conta aqui é parecida com a que fazemos com base decimal (que é a base que usamos, ou seja, representando todos os números possíveis usando apenas de 0 a 9). Veja que não temos um "desenho" para o número dez. O que fazemos quando chegamos no maior número da nossa escala (que no caso é 9)? Empurramos um dígito para esquerda, e começamos a contagem de novo: 10, 11, 12, 13, e assim por diante.

Com números binários, é a mesma coisa. Tínhamos o 0, e depois o 1. Como 1 é o maior dígito que temos, empurramos um dígito para a esquerda e começamos a contagem de novo: 0, 1, 10, 11, 100, 101, 110, 111, e assim por diante.

Assim como na matemática tradicional, podemos preencher com zero à esquerda para padronizar a quantidade de dígitos. Por exemplo, se usarmos 8 dígitos ou seja, 8 bits , temos:

```
0  =>  00000000
 1  =>  00000001
 2  =>  00000010
 3  =>  00000011
 4  =>  00000100
```

```
 5 => 00000101
 6 => 00000110
 7 => 00000111
 8 => 00001000
 9 => 00001001
10 => 00001010
11 => 00001011
12 => 00001100
13 => 00001101
14 => 00001110
15 => 00001111
```

O conjunto de 8 bits juntos é conhecido por 1 byte. Podemos seguir o mesmo padrão até qualquer número inteiro. Um inteiro em C tem 4 bytes. Isso quer dizer que usamos `4 vezes 8` = 32 bits. Com 32 dígitos, conseguimos representar números bastante grandes.

10.1 Binário e letras

Mas como representar letras (chars)? Basta fazermos uma tabela, dando um número para cada letra. Por exemplo, 'A = 65'. Então 'B = 66', 'C = 67':

```
A => 65 => 01000001
B => 66 => 01000010
C => 67 => 01000011
...
Z => 90 => 01011010
```

Pronto, o computador é capaz de representar todas as letras existentes no nosso alfabeto usando somente 8 dígitos, 0 ou 1. Essa tabela do nosso alfabeto, incluindo diversos outros caracteres é a **tabela ASCII**, usada por muito tempo como o principal padrão de tradução de números e caracteres por um computador.

10.2 Bits: 8, 16, 32, 64

Se tivéssemos apenas 8 bits para representar um número inteiro, poderíamos representar `2 elevado a 8` números, que é igual a 256. Parece pouco,

mas acredite ou não, nossos primeiros videogames, como o Master System, tinham processadores de 8 bits e sabiam só representar números de 0 a 255.

O Mega Drive tinha um processador um pouco melhor de 16 bits, e com isso, representava números inteiros de 0 a 65.535 (ou seja, `2 elevado a 16` números diferentes). Não é à toa que os jogos do Master tinham, em geral, no máximo 256 cores, enquanto as do Mega tinham até 65 mil.

Mas mesmo esse número, será que ele é capaz de representar todos os números inteiros que queremos? Ou ainda um processador capaz de entender e armazenar 16 bits também está apto para representar todas as letras de todos os alfabetos do mundo?

Os processadores modernos são capazes de representar números com até 64 bits o que dá bons bilhões , além de todo o conjunto de caracteres dos alfabetos existentes no mundo. Mesmo assim, existem otimizações para representar o alfabeto, como um padrão famoso por ter substituído o ASCII, sendo o mais utilizado na internet em 2008, o UTF-8.

10.3 Bits e números com ponto flutuante

Agora temos a questão dos números com ponto flutuante. Como representar tais números, se o computador só conhece zeros e uns? Do mesmo jeito que fizemos com as letras, precisamos de algum padrão. Para entender melhor, vamos criar o nosso próprio padrão, mais simples do que um usado hoje em dia.

Imagine o número treze e meio (13,5). Quantos dígitos ele possui antes da casa decimal? Dois (em binário, `0010`). Quais são esses dígitos? `1` (`0001`) e `3` (`0011`). Por fim, o valor da casa decimal é `5` (`0101`). Portanto, para representar o número `13,5`, utilizaremos a notação que indica o número de casas antes da vírgula (`2`), o seu valor dígito a dígito (`1` e `3`) , e o valor decimal (`5`):

```
0010 0001 0011 0101
```

Pronto. O computador é capaz de entender números como `13,5`. Com isso, somos capazes de representar qualquer número com casas decimais, cuja quantidade de dígitos é finita, mesmo que essa técnica esteja longe de ser ótima ou perfeita.

Entretanto, ainda temos uma limitação: neste tipo de abordagem, como representar o valor `1 / 3`, isto é, um terço? Esse número tem uma quantidade infinita de dígitos.

Uma primeira opção seria arredondarmos o número `0,3`, perdendo precisão:

```
0001 0000 0011
```

Já que o arredondamento foi ruim, podemos tentar melhorá-lo `0,33`:

```
0001 0000 0011 0011
```

Ou mais ainda, `0,333`:

```
0001 0000 0011 0011 0011
```

Como o computador costuma seguir padrões de tamanho fixo, é comum que o número de ponto flutuante, assim como um número inteiro, utilize 64 bits. Isso limita a nossa aproximação. Para nosso um terço, usando essa ideia apresentada (não ideal), temos a melhor aproximação de `0,333.3` com `62` casas após a vírgula.

```
0001 0000 0011 0011 0011 ... 0011
```

Nossa abordagem é bastante limitada, mas a representação escolhida por um processador é bem mais inteligente que essa e permite aproximações muito boas.

10.4 Hexadecimal

A base binária, como vimos, tem somente dois algarismos distintos para representar *todos* os números. A que usamos no dia a dia tem dez algarismos. Entretanto, há uma outra base bastante utilizada, a base `hexadecimal`, que tem 16 algarismos. Para representar números de 0 a 15, precisamos apenas de 4 bits.

Agora precisamos de 16 "desenhos" para representar os 16 possíveis algarismos. Para isso completamos os números de 0 a 9 com as letras A, B, C, D,

E e F para representar os outros seis algarismos faltantes. Novamente, sem a necessidade de decorar a fórmula de transformação, a tabela a seguir mostra os 16 primeiros números em base hexadecimal:

```
0 => 0
 1 => 1
 2 => 2
 3 => 3
 4 => 4
 5 => 5
 6 => 6
 7 => 7
 8 => 8
 9 => 9
10 => A
11 => B
12 => C
13 => D
14 => E
15 => F
```

Poderíamos continuar com:

```
16  => 10
17  => 11
18  => 12
...
32  => 21
...
177 => B1
...
255 => FF
```

Portanto, com dois algarismos de base hexadecimal (16 possibilidades cada), conseguimos representar 16 * 16, 256, números.

10.5 Bits e imagens

Agora, um último desafio: como representar uma imagem? É fácil. Mapearemos uma imagem para um conjunto de números inteiros. Já sabemos que

conseguimos representar números inteiros de forma binária.

Imagine uma foto. Ela é composta por diversos pontos, sendo que cada um deles tem uma cor diferente. Suponha que a foto tem 1000 por 1000 pontos. O que podemos fazer é primeiro enumerar todas as cores existentes. Mas vamos limitá-las a 256 diferentes:

```
cor0   = verde
cor1   = vermelho
cor2   = azul
cor3   = preto
cor4   = branco
...
cor255 = cinza
```

Não podemos usar o nome da cor: precisamos dar um número a ela. Por exemplo:

```
cor0   = 00FF00 (verde)
cor1   = FF0000 (vermelho)
cor2   = 0000FF (azul)
cor3   = 000000 (preto)
cor4   = FFFFFF (branco)
...
cor255 = CCCCCC (cinza)
```

Com essa tabela em mãos, podemos escrever uma linha para cada ponto na foto. Se o primeiro ponto é vermelho (cor1), o segundo azul (cor2) e o terceiro cinza (cor255), em uma imagem com diversas linhas, temos:

```
cor1 cor2 cor255 cor3 cor3 cor2 cor3 cor255
cor1 cor3 cor255 cor3 cor2 cor3 cor3 cor255
...
```

Mas como sabemos que temos, no máximo, 256 cores, podemos representá-las mais facilmente com números em hexadecimal:

```
01 02 FF 03 03 02 03 FF
01 03 FF 03 02 03 03 FF
...
```

Isto é, a sequência `01 02 FF` representa uma pequena imagem de 3 pontos, cujas cores seriam vermelho, azul e cinza. Já `0102FF03030203FF\n0103FF03020303FF` representa uma imagem com 2 linhas e 8 colunas.

Com isso, podemos representar qualquer imagem com tamanho finito (que caiba na memória) e cujo número de cores esteja limitado a `256`. Esse é o raciocínio por trás de um mapa de bits de cores, um *bitmap* o formato `BMP` , muito utilizado antigamente no mundo da computação. Sua adoção não é mais tão grande devido ao tamanho que ele ocupa: uma imagem com grande qualidade exige uma tabela de cores enorme, e um número de pontos maior ainda. Algoritmos de compressão como o `JPEG` dominam esse mercado.

Assim como pensamos na alternativa de representar um terço como uma fração (um dividido por três), poderíamos representar uma imagem como uma sequência de traços, círculos etc. Essa também é uma ideia boa para diversos tipos de imagem, e é o raciocínio por trás das imagens vetoriais, como o *SVG*.

O mesmo pode ser aplicado ao mundo da música, com o `WAV` e `MP3`, aos filmes etc. No fim, representamos desde um número inteiro até uma imagem de tomografia com apenas zeros ou uns. Assustador.

10.6 Resumindo

Neste capítulo, aprendemos:

- Que computadores conseguem representar apenas 0s e 1s na memória;
- Que conseguimos escrever qualquer número com números binários;
- Que podemos criar padrões e regras para escrever números com ponto flutuante, usando números binários;
- Que números hexadecimais também são bastante importantes e utilizados;
- Como funciona o padrão *bitmap*.

Capítulo 11

Funções e ponteiros

Você reparou que nosso jogo ainda não tem uma mensagem de boas-vindas ao usuário? Vamos colocá-la:

```
int main() {

    // imprime cabeçalho
    printf("/****************/\n");
    printf("/ Jogo de Forca */\n");
    printf("/****************/\n\n");

    // código continua aqui
}
```

Nosso jogo, em termos de funcionalidades, está andando bem. Mas vamos dar uma pausa nelas, para atacar outro detalhe importantíssimo na hora

de fazermos um software. Você se lembra do código do jogo de adivinhação? Ele ficou muito legal, porém, ao final da implementação, tínhamos uma função `main()` com mais de 100 linhas de código. São muitas linhas, e isso dificulta muito o entendimento do código por outras pessoas. Faça o teste você mesmo. Daqui a algumas semanas, volte e olhe o mesmo código: você lembrará muito menos dele.

Precisamos aprender a dividir nosso programa em partes menores, que sejam mais fáceis de serem lidas. É necessário fazermos igual as bibliotecas do C fazem. Por exemplo, quando você quer imprimir algo na tela, você chama a função `printf()`. Você não sabe como ela funciona por dentro, apenas sabe que ela imprime. A mesma coisa com o

11.1 Escrevendo funções

Imagine se conseguíssemos trocar as 3 linhas responsáveis pela abertura do jogo por uma única? Algo parecido com o comentário, mas em formato de código, sem espaços, usando os parênteses e terminando a linha com ponto e vírgula:

```
abertura();
```

O que queremos aqui é criar o que chamamos de **funções**. Funções são blocos de código encapsulados e que podem ser reutilizados ao longo do nosso programa. Por exemplo, se criarmos uma função que se chama `abertura()` contendo aquelas 3 linhas de impressão , sempre que a invocarmos, nosso programa fará isso. É igual ao `printf()` ou ao `scanf()`: ambos são funções que usamos ao longo do nosso programa.

Vamos criar essa função e usá-la. Escreveremos esse código fora da função `main`; afinal, ela própria é uma função, e não podemos ter funções dentro de funções. Para declará-la, precisamos no mínimo de um nome. Pode ser qualquer palavra. Faremos `void abertura()`. Você não precisa entender o `void` por enquanto, e nem o abre-e-fecha parênteses. Em seguida, abriremos chaves e colocaremos o conteúdo dela dentro desse bloco de código:

```
void abertura() {
    printf("/****************/\n");
```

```
    printf("/ Jogo de Forca */\n");
    printf("/****************/\n\n");
}
```

Então, vamos fazer uso dessa função. Isso é feito dentro da função `main()`. Vamos remover aquelas 3 linhas, que agora estão já na função certa, e apenas invocá-la. Para invocar é fácil, usamos seu nome e parênteses:

```
int main() {

    abertura();

    // código continua aqui

}
```

Se rodarmos nosso código, o comportamento ainda é o mesmo. A diferença é que agora nossa função `main` é um pouco mais fácil de ser lida. Quando o programador lê `abertura()`, ele sabe que é ali que é feita a abertura, mas não sabe os detalhes. Se quiser saber, ele deve ler o corpo da função `abertura()`. Diminuímos a quantidade de código que o programador precisou ler para entender a função `main()`.

Vamos entender o que acontece com a máquina ao executar esse código. Quando o programa começa, ele automaticamente executa a função `main()` e, depois, linha a linha do código. Até aí, sem novidades. Porém, quando ele encontra uma chamada para outra função por exemplo, `abertura()` , ele "suspende" a execução de `main()` e vai para a outra. Lá, o procedimento é o mesmo. Ele executa linha a linha e, quando a função `abertura()` acaba, ele volta para `main()` e continua a execução a partir dali.

Se tivéssemos uma outra chamada de função dentro de `abertura()`, o comportamento seria o mesmo. A máquina suspenderia a execução desta, executaria a outra e, quando ela acabasse, voltaria para a anterior, que, por sua vez, quando acabasse, voltaria para a função `main()`.

Está perdido?

O código até esse momento está em http://pastebin.com/ES8eiBdf

11.2 Mais funções

Veja agora o código principal do nosso jogo e, mais especificamente, o conteúdo dentro do `do-while`. Repare nos três comentários de código. Será que também não conseguimos transformá-los em funções?

```
do {
    // imprime a palavra secreta
    for(int i = 0; i < strlen(palavrasecreta); i++) {
        int achou = 0;

        // a letra já foi chutada?
        for(int j = 0; j < tentativas; j++) {
            if(chutes[j] == palavrasecreta[i]) {
                achou = 1;
                break;
            }
        }

        if(achou) {
            printf("%c ", palavrasecreta[i]);
        } else {
            printf("_ ");
        }
    }
    printf("\n");

    // captura um novo chute
    char chute;
    printf("Qual letra? ");
    scanf(" %c", &chute);
```

```
    chutes[tentativas] = chute;
    tentativas++;


} while (!acertou && !enforcou);
```

Repare no primeiro comentário: *imprime a palavra secreta*. Ele engloba o `for`, que é responsável por imprimir a forca. O segundo comentário, *a letra já foi chutada?*, engloba o segundo `for`, que é responsável por dizer se uma determinada letra já foi chutada pelo usuário ou não. Por fim, o último comentário, *captura um novo chute*, pega um novo chute do usuário.

Vamos pegar o último comentário e também transformá-lo em uma função. Afinal, isso deixará nosso código ainda mais legível. O procedimento é o mesmo. Logo, vamos criar a função `chuta()`:

```
void chuta() {
    char chute;
    printf("Qual letra? ");
    scanf(" %c", &chute);

    chutes[tentativas] = chute;
    tentativas++;
}
```

Vamos agora também invocá-la dentro do `main()`, no lugar do código antigo que removemos em prol dessa nova função:

```
int main() {
    do {
        // ...

        chuta();
    } while(!acertou && !enforcou);
}
```

Fizemos o mesmo procedimento de anteriormente, mas, dessa vez, o código não compila. Veja o erro: ele reclama que a variável `chutes` não existe. E o compilador está certo; afinal, a variável `chutes` existe apenas dentro da função `main()`:

```
forca.c:15:2: error: use of undeclared identifier 'chutes';
did you mean 'chute'?
        chutes[tentativas] = chute;
        ^~~~~~
        chute
```

11.3 Passando parâmetros para funções

Variáveis vivem, e são visíveis, apenas no escopo em que foram declaradas. Por exemplo, se declaramos uma variável dentro da função `main()`, ela estará visível somente dentro dessa função e dos escopos declarados dentro dela. Por exemplo, se você tiver um `for` dentro dela, ela conseguirá ver essa variável.

No código a seguir, a variável `multiplicador`, declarada dentro da função `main()`, é visível em todo lugar dentro dessa função. Já a variável `resultado`, declarada dentro do `for`, é visível somente dentro desse `for` (ou em outros escopos que fossem declarados dentro dele). Entretanto, nenhuma delas é visível dentro da função `abertura()`:

```
void abertura() {
    // aqui não enxergamos a variável multiplicador
    printf("Tabuada do ");
    printf("Quero imprimir o multiplicador aqui,
            mas nao consigo...");
}

int main() {
    int multiplicador = 2;

    abertura();

    for(int i = 0; i < 10; i++) {
        int resultado = multiplicador * i;
        printf("%d x %d = %d", i, multiplicador,
            resultado);
    }
}
```

Para resolver esse problema, temos duas soluções. A primeira delas é passar as variáveis que queremos para as outras funções. Isso pode ser feito; basta, no momento da declaração da variável, dizer quais serão os parâmetros que ela receberá. Na hora de invocarmos, passamos os parâmetros para ela nos "parênteses" no momento da declaração. Basta dizer o nome do parâmetro e seu tipo.

Veja, por exemplo, a função `abertura()` recebendo como parâmetro um inteiro `multiplicador`. Esse parâmetro é como se fosse uma variável para aquela função. Ela existe naquele escopo e pode ser manipulada como qualquer outra:

```
void abertura(int multiplicador) {
    // aqui temos uma variável chamada multiplicador
    // e seu valor será definido por quem chamar essa função
    printf("Tabuada do %d", multiplicador);
}
```

Isso significa que não podemos mais invocar a função `abertura()` sem passar os seus parâmetros. Sempre precisamos passar os parâmetros que as funções requerem. Nesse caso em particular, precisamos sempre passar um inteiro para ela. Observe:

```
// passando um inteiro diretamente
abertura(2);
```

Já que ela recebe um inteiro, por que não passar também um inteiro que está em uma variável?

```
int main() {
    // passando uma variável
    int numero = 10;
    abertura(numero);
}
```

Repare que estamos passando a variável `numero` como parâmetro para a função `abertura()`. Ela contém o número 10. Isso quer dizer que, quando a função `abertura()` for executada, o parâmetro `multiplicador` valerá

10 também. É como se a máquina copiasse o valor da variável `numero` e colasse na variável `multiplicador`.

Para que esse nosso pequeno código de tabuada funcione, basta invocarmos a função `abertura()`, passando a variável `multiplicador`. Note que, apesar de ambos os nomes serem iguais (o da variável e o do parâmetro), não é o nome que fez tudo acontecer. Ou seja, o parâmetro `multiplicador` é diferente da variável `multiplicador` que está na `main()`; apenas os conteúdos é que são iguais. Lembre-se de que, quando você chama uma função, por exemplo, `abertura(10)`, a máquina pega o número 10 e coloca dentro do parâmetro que foi declarado na função (que pode ter qualquer nome).

```
void abertura(int multiplicador) {
    printf("Tabuada do %d\n", multiplicador);
}

int main() {
    int multiplicador = 2;

    abertura(multiplicador);

    for(int i = 0; i < 10; i++) {
        int resultado = multiplicador * i;
        printf("%d x %d = %d\n", i, multiplicador,
            resultado);
    }
}
```

Está perdido?

O código do exemplo da tabuada está em http://pastebin.com/zAWw0wbF

Voltando ao nosso jogo, precisamos fazer com que a função `chuta()` receba o array de chutes e a variável `tentativas`. Vamos colocar esse parâmetro na assinatura da função. Repare como declaramos que o parâmetro `chutes` é do tipo array de char.

```
void chuta(char chutes[], int tentativas) {
    char chute;
    printf("Qual letra? ");
    scanf(" %c", &chute);

    chutes[tentativas] = chute;
    tentativas++;
}
```

Agora, passamos o array e o inteiro, que foram declarados na função `main`, para a função `chuta()`:

```
chuta(chutes, tentativas);
```

Novamente, note que os parâmetros `chutes` e `tentativas` são "variáveis diferentes" do `chutes` e `tentativas` que estão na função `main()`. Porém, isso pode ser um problema, dependendo do caso. Falaremos mais sobre isso em breve.

Está perdido?

O código até esse momento está em http://pastebin.com/N2QMxqfE

11.4 Ponteiros

Veja o programa a seguir. Nele queremos provar que, quando usamos parâmetros em funções e passamos uma variável como parâmetro, mesmo que os nomes delas sejam iguais, elas ainda são diferentes. A função `main()` contém uma variável `tentativas`, faz operações de incremento com esta e imprime. Em seguida, ela a passa para a função `joga()`, que faz um incremento, e imprime de novo. Ao final, a função `main()` imprime novamente o valor da variável.

```
#include <stdio.h>

void joga(int tentativas) {
```

```
    printf("joga %d\n", tentativas);
    tentativas++;
    printf("joga %d\n", tentativas);
}

int main() {

    int tentativas = 0;
    printf("main %d\n", tentativas);

    tentativas++;
    printf("main %d\n", tentativas);

    tentativas++;
    printf("main %d\n", tentativas);

    joga(tentativas);

    printf("main %d\n", tentativas);

}
```

O interessante é a saída desse programa. Veja que a variável na função `main()` vale 2, quando chamamos a função `joga()`. Em seguida, a função `joga()` incrementa 1, e podemos ver que ela imprime o número 3. Mas, quando voltamos para a função `main()`, ele imprime 2 novamente:

```
main 0
main 1
main 2
joga 2
joga 3
main 2
```

Isso nos prova que ambas as variáveis, apesar de terem o mesmo nome, são diferentes. Quando invocamos a função `joga()`, passando a variável `tentativas` declarada na função `main()`, a máquina copia seu conteúdo e cria uma nova variável chamada `tentativas`, que será visível só na função

`joga()`. Quando a função `joga()` termina, e o programa volta para a função `main()`, a variável `tentativas` continua valendo 2. Afinal, ela não foi modificada depois disso.

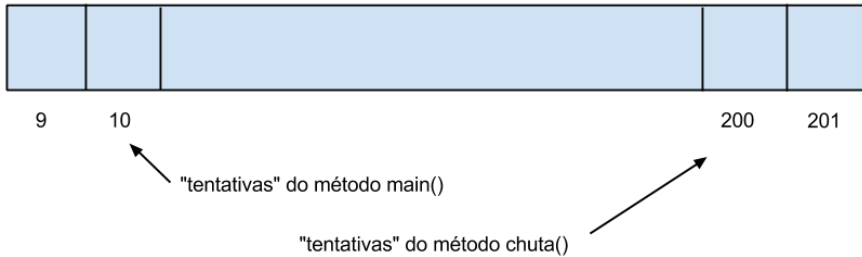



Fig. 11.1: Variáveis com mesmo nome em regiões diferentes da memória

Esse isolamento entre as funções é uma coisa boa. Imagine se você passasse uma variável como parâmetro para uma função, e esta tivesse a liberdade de mudá-la? Seu programa seria mais complicado de ser entendido; afinal, você não saberia quem mudou a variável e quando isso aconteceu. É por esse motivo que chamamos isso de **passagem por cópia**. Ou seja, sempre que passamos uma variável para uma função, a máquina a copia e cria uma nova, para que a original fique intacta.

Muitas vezes a passagem por cópia é suficiente. Mas às vezes, não. Volte ao nosso jogo e veja a variável `tentativas`. Ela contém a quantidade de chutes já dados pelo jogador. A função `chuta()` a incrementa. Mas, na verdade, deveríamos incrementar a variável `tentativas` que está dentro do escopo da função `main()`. Nesse momento, o parâmetro `tentativas` da função `chuta()` é o que é incrementado, e não o da função

E para resolver isso? Gostaríamos, nesse caso, que a variável `tentativas` da função `chuta()` fosse a mesma da função `main()`.

```
void chuta(char chutes[26], int tentativas) {
    char chute;
    scanf(" %c", &chute);

    chutes[tentativas] = chute;
```

```
    // a soma precisa acontecer na variável
    // 'tentativas', que existe no escopo
    // da função 'main'
    tentativas++;
}

int main() {
    int tentativas = 0;

    // ...

    chuta(chutes, tentativas);

    // ...
}
```

11.5 Passagem por referência

Podemos dizer ao nosso programa que não queremos que ele passe o parâmetro por cópia, mas sim, pelo que chamamos de **referência**. A passagem por referência é algo bastante avançado. Como que a máquina faz para passar uma variável por referência? Simples, ela passa a posição da memória em que essa variável está. Imagine a memória do computador como uma fita imensa e numerada. Todas as variáveis de nossos programas ficam em algum lugar dessa fita.

Pense novamente na passagem por cópia. Imagine que a variável `tentativas` da função `main()` está na posição 10 da memória. Quando passamos por cópia, a máquina aloca um novo espaço na memória para a nova variável `tentativas`. Suponha que ela ficou na posição 200. Ou seja, uma variável aponta para a posição 10, enquanto a outra aponta para a posição 200. Você precisa pensar agora em variáveis como se elas fossem setas, ou melhor, **ponteiros** para endereços de memória, pois é exatamente isso o que elas são. Veja na figura a seguir as duas variáveis que têm o mesmo nome `tentativas`, mas se encontram em regiões diferentes da memória:

Mas, se no fundo, variáveis são apenas ponteiros para endereços de me-

mória, por que não ter duas variáveis que apontam para o mesmo endereço? Podemos fazer isso. Por exemplo, se quisermos ver o endereço de memória que uma variável aponta, usamos o caractere `&`. O programa a seguir imprime o endereço de memória da variável `c`. Repare que, sempre que o rodarmos, o número será diferente; afinal, cada hora a máquina decide pegar um endereço de memória diferente que está livre naquele momento, claro:

```
#include <stdio.h>

int main() {

    int c = 10;
    printf("%d\n", &c);

}
```

Isso quer dizer que, se passarmos como parâmetro para uma função um `&c`, por exemplo , estaremos passando o endereço de memória em que essa variável se encontra. Ou seja, caso mudarmos o conteúdo que está dentro desse endereço, ele será modificado para todo mundo que tenha uma variável apontada para lá.

Se você declara uma variável do jeito que está acostumado, a máquina esconde de você a ideia de que ela é, na verdade, um ponteiro. É por isso que você consegue fazer coisas como atribuições por exemplo, `c = 10` , e a máquina faz o processo de pegar o endereço de memória que está escondido em `c` e colocar o valor 10.

Você também tem a opção de manipular diretamente um endereço de variável, ou seja, declarar um ponteiro. Um ponteiro simplesmente contém um endereço de memória. Declará-lo é parecido com a declaração de uma variável: precisamos de um nome e de um tipo. Mas completamos o tipo com estrela ou asterisco (`*`).

Veja o exemplo a seguir, no qual declaramos um ponteiro para um inteiro. Se ele é um ponteiro, quer dizer que podemos guardar o endereço da variável `c` lá dentro. Para pegar o endereço dela, basta usarmos o `&`.

```
int c = 10;
```

```
int* ponteiro;
ponteiro = &c;
```

Nesse momento, a variável `ponteiro` aponta para o mesmo lugar ao qual `c` aponta. Se imprimirmos `&c` e `ponteiro`, o resultado é o mesmo:

```
printf("%d %d", ponteiro, &c);
```

Agora, se quisermos acessar o que está dentro do endereço para qual `ponteiro` aponta, usamos o `*` na frente da variável. Por exemplo, se imprimirmos a variável `c` e `*ponteiro`, o resultado é o mesmo (10):

```
printf("%d %d", *ponteiro, c);
```

Ou seja, se a variável é um ponteiro, e queremos acessar o conteúdo que está dentro do endereço ao qual ela aponta, usamos estrela. Veja novamente o código inteiro e perceba como manipulamos uma variável normal e uma variável ponteiro:

```
int c = 10;
int* ponteiro;

// ponteiro apontando para o mesmo endereço de c
ponteiro = &c;

// imprime o endereço da variável c
printf("%d %d\n", ponteiro, &c);

// imprime o conteúdo da variável c
printf("%d %d\n", *ponteiro, c);
```

Sabendo disso, vamos passar a variável `tentativas` como referência. Ou seja, vamos passar seu endereço de memória para a função `chuta()`, usando o `&`:

```
chuta(chutes, &tentativas);
```

Claro que, se passamos um ponteiro de inteiro para a função, é porque ela recebe um ponteiro de inteiro. Precisamos mudar a assinatura da função `chuta()`, e fazê-la receber um `int* tentativas`. Além disso, devemos usar o `*` para acessar o conteúdo que o ponteiro `tentativas` aponta. Veja:

```
void chuta(char chutes[], int* tentativas) {
    char chute;
    printf("Qual letra? ");
    scanf(" %c", &chute);

    chutes[*tentativas] = chute;
    (*tentativas)++;
}
```

Podemos agora colocar um `printf()` em nosso jogo, mostrando o número de tentativas. Você verá que, como estamos passando o ponteiro da variável, o número é modificado na variável que queremos.

```
do {
    printf("Você já deu %d chutes\n", tentativas);

    // ...
}
```

Ponteiros, apesar de parecerem complicados em um primeiro instante, são poderosíssimos. Na figura a seguir, veja que agora ambos os ponteiros `tentativas` apontam para o mesmo endereço de memória, logo possuem o mesmo conteúdo:

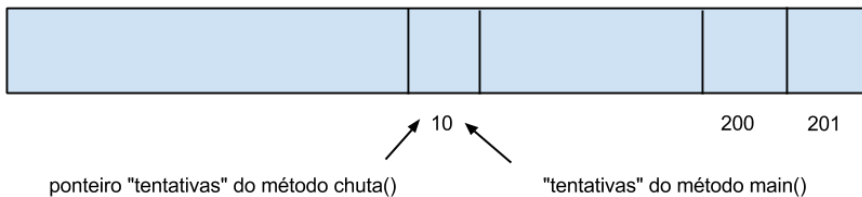



Fig. 11.2: Funcionamento dos ponteiros

Está perdido?

O código feito até o momento está em http://pastebin.com/ztjPwSvY

11.6 Arrays e ponteiros

Você deve estar se perguntando o motivo pelo qual tivemos que passar o inteiro como ponteiro, mas não o array. Isso não muda: se quisermos modificar uma variável que está declarada em outro escopo, precisamos mexer direto em seu endereço de memória, e usar ponteiros; não tem solução.

Acredite ou não, fizemos isso no array. Um array é, por padrão, um ponteiro. A variável `chutes` é um ponteiro que aponta para o inteiro que está na primeira posição do array. Sabemos que a primeira posição do array é `chutes[0]`. Se fizermos `&chutes[0]`, temos o endereço de memória desse primeiro inteiro. Esse número é igual a `chutes`.

Veja o código:

```
printf("%d %d", chutes, &chutes[0]);
```

Agora que você está entendendo mais sobre a memória, veja que um array é uma "sequência de variáveis" declaradas. Em particular, essas variáveis estão exatamente uma do lado da outra. Quando declaramos um array de 10 posições, a máquina procura por um lugar onde caibam 10 inteiros, um ao lado do outro.

Quando fazemos `chutes[2]`, por exemplo, a máquina pega a posição de memória da primeira posição desse array, e vai navegando até a posição que quer. No exemplo, o número 2 quer dizer a terceira posição do array. Ele pula 3 casas e chega ao próximo inteiro.

Mas você se lembra que discutimos o tamanho de um inteiro? Um inteiro ocupa 4 bytes. Em outras palavras, se você declara uma variável inteira, a máquina separa 4 bytes da memória para ele. Em um array, acontece a mesma coisa; se temos um array de inteiros de 10 posições, precisamos de 40 bytes. Ou seja, se `&chutes[0]` nos retorna 10, `&chutes[1]` nos retornará 14,

pois é onde o segundo inteiro começa. Veja na figura a seguir um diagrama do funcionamento de um array e como lidamos com ele:

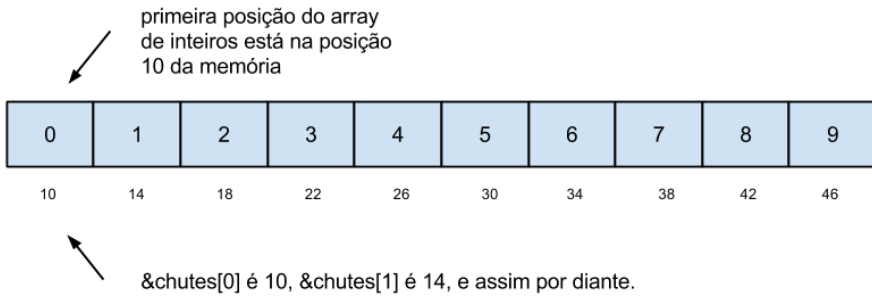



Fig. 11.3: Funcionamento de um array

Veja em código. O programa a seguir declara um array de inteiros de 3 posições. Em seguida, imprimimos a posição de memória em que estão guardados cada um desses números. Sabemos que um inteiro tem 4 bytes e os itens de um array são guardados um ao lado do outro. Portanto, os números devem crescer de 4 em 4.

```
#include <stdio.h>

int main() {
    int numeros[3];

    printf("%d %d %d\n", &numeros[0], &numeros[1], &numeros[2]);
}
```

A saída com os endereços é diferente a cada vez que executarmos o trecho de código, pois o endereço de memória muda. Note que ela pula de 4 em 4 bytes, como esperado:

```
1458119804 1458119808 1458119812
```

Podemos tratar esse `int[]` como um `int*`. Afinal, arrays são ponteiros. Se temos um ponteiro de inteiro e sabemos que ele aponta para um array, podemos acessar a primeira posição fazendo `*ponteiro`. Se quisermos acessar

a segunda posição, basta somarmos 1 a esse ponteiro, `*(ponteiro + 1)`, e assim por diante.

Observe o código a seguir, no qual fazemos um loop e acessamos os valores do array, usando o tipo array convencional e também diretamente o ponteiro:

```
#include <stdio.h>

int main() {
    int numeros[3];

    // declarando um ponteiro que aponta para o
    // mesmo lugar que o array/ponteiro numeros
    int* ponteiro = numeros;

    numeros[0] = 10;
    numeros[1] = 20;
    numeros[2] = 30;

    for(int i = 0; i < 3; i++) {
        printf("%d ", numeros[0]);

        // repare na soma que fazemos com o ponteiro
        printf("%d ", *(ponteiro + i));

        printf("\n");
    }
}
```

A partir de agora, sua cabeça deve estar muito diferente. Você consegue ver ponteiros em tudo? Lembre-se de que variáveis são ponteiros para endereços de memória. A linguagem de programação geralmente nos esconde isso. Ela se vira em manipular esse endereço e salvar nossos dados lá. Entretanto, se precisarmos, a linguagem C nos deixa lidar diretamente com esses endereços de memória.

11.7 Funções com retorno

A próxima função que vamos extrair de nosso código será responsável por nos dizer se o jogador já chutou determinada letra. Veja que é isso que o `for` de dentro faz. Ele varre a palavra secreta e a lista de chutes, e guarda na variável `achou` se o usuário já chutou ou não aquela letra.

É fácil ver que esse `for` usa o array de chutes; afinal, ele necessita passar por todo ele, e precisa também da letra que será procurada. Vamos escrever a função `void jachutou()`, que receberá um `char letra`, um `char* chutes` lembre-se que um array é um ponteiro, então podemos receber como um ponteiro , e a quantidade de tentativas.

Vamos também relembrar a lógica dessa função. Ela varria o array, procurando por um chute que seja igual à letra com qual ele quer comparar. Se ele achar a letra no array, ele marca a variável `achou` como 1. O 1, para nós, significa "verdadeiro". Caso contrário, `achou` é falso. Essa variável, então, era usada mais para baixo, para exibir ou não a letra. Entretanto, isso não vem ao caso, já que não faz parte do processo de dizer se uma letra já foi ou não chutada.

```
void jachutou(char letra, char* chutes, int tentativas) {
    for(int j = 0; j < tentativas; j++) {
        if(chutes[j] == letra) {
            achou = 1;
            break;
        }
    }
}
```

Com a função escrita, vamos usá-la e invocá-la no lugar onde o código antigo existia. O código ficou simples. Declaramos a variável `achou`, inicializando-a com 0, e invocamos nossa função `jachutou`. Em seguida, o `if`, que imprime ou não a letra, usa essa variável:

```
for(int i = 0; i < strlen(palavrasecreta); i++) {
    int achou = 0;

    jachutou(palavrasecreta[i], chutes, tentativas);
```

```
    if(achou) {
        printf("%c ", palavrasecreta[i]);
    } else {
        printf("_ ");
    }
}
```

Porém, esse código ainda não funciona. Afinal, já sabemos que a variável `achou`, declarada dentro da função `main()`, não é a mesma da declarada dentro da `jachutou()`. Se quisermos "compartilhar" a variável, precisaremos passar o seu ponteiro. Sabemos que isso não é difícil, pois bastaria passar o endereço de `achou`, usando `&achou`, e receber o ponteiro do outro lado, como parâmetro, um `int* achou`.

Apesar de a alternativa ser válida, geralmente temos uma solução mais elegante para isso. Veja que a variável `achou` é o resultado da nossa função `jaachou()`. Ou seja, ela processa, faz um monte de coisa, mas, ao final, precisa apenas "contar para quem a chamou" o resultado dessa variável. Quando uma função precisa devolver apenas um dado um número, ou char, ou double etc. para quem a chamou, damos um tipo de retorno a ela.

Até agora, sempre que declaramos uma função, usamos a palavra `void` no começo. Usamos `void` quando a função não precisa retornar nada a quem a chamou. Mas, se ela precisar retornar, podemos trocar essa palavra pelo tipo de retorno. Por exemplo, queremos devolver um inteiro, então trocamos a palavra `void` por `int`:

```
int jachutou(char letra, char* chutes, int tentativas) {
    // ...
}
```

Agora, precisamos indicar qual o valor que deve ser retornado por essa função. Para fazer isso, basta usar a palavra `return`. Quando esse termo aparece dentro de uma função, geralmente acompanhado por um valor por exemplo, `return 10;` para retornar o valor 10 , a função para ali, e quem a chamou recebe esse valor.

Aqui, queremos retornar o conteúdo da variável `achou` ao final da função. Colocaremos como última linha esse retorno. Além disso, precisamos

declarar a variável `achou` no começo dessa função e inicializá-la com 0, pois ela não existia ali até então:

```
int jachutou(char letra, char* chutes, int tentativas) {
    int achou = 0;

    for(int j = 0; j < tentativas; j++) {
        if(chutes[j] == letra) {
            achou = 1;
            break;
        }
    }

    return achou;
}
```

Nossa função `jachutou()` está pronta. Ela nos devolve um inteiro sempre que a invocarmos. Esse inteiro vale 1 quando ele achar a letra, e 0 quando não achar. Veja que isso parece uma função da matemática. Dado um conjunto de valores de entrada, ela nos devolve um resultado como saída. É por isso que chamamos de função.

Como a matemática nos dá bons exemplos de funções, poderíamos ter uma que nos devolvesse o quadrado de um número. Ela receberia um número como parâmetro e devolveria a multiplicação dele por ele mesmo. Veja em código:

```
int quadrado(int n) {
    return n * n;
}
```

Usar essa função é igual usar funções que devolvem `void`. A diferença, claro, é que elas nos retornam algo. Para capturarmos o valor retornado, precisamos guardá-la em uma variável. Veja alguns exemplos de uso da função `quadrado()`; umas guardando em variáveis, outras usando-as diretamente:

```
// guardando em uma variável
int resultado = quadrado(2);
printf("%d", resultado);
```

```
// usando diretamente em um if
if(quadrado(3) < 10) {
    printf("Já sabia, o quadrado de 3 é 9");
}

// usando direto em um printf
printf("%d", quadrado(5));
```

Voltando ao jogo, vamos agora fazer uso direto da função `jachutou()`. Como vimos que podemos utilizá-la diretamente dentro do `if`, sem a necessidade de uma variável auxiliar, vamos usá-la diretamente no condicional que decide mostrar ou não a letra. Veja como nosso código está cada vez mais simples:

```
for(int i = 0; i < strlen(palavrasecreta); i++) {

    if(jachutou(palavrasecreta[i], chutes, tentativas)) {
        printf("%c ", palavrasecreta[i]);
    } else {
        printf("_ ");
    }

}
```

Está perdido?

O código até o momento está em http://pastebin.com/G1Lp35mS

11.8 Extraindo mais funções

Vamos continuar separando em funções. Agora, podemos extrair o loop `for` de fora, aquele que varre a palavra secreta, para uma função `desenhaforca()`. Extraí-la é fácil. Basta fazermos a função receber a palavra secreta, os chutes e a quantidade de tentativas. Como ela não precisa retornar nada, ela é `void`:

```
void desenhaforca(char* palavrasecreta, char* chutes,
                  int tentativas) {

    printf("Você já deu %d chutes\n", tentativas);

    for(int i = 0; i < strlen(palavrasecreta); i++) {

        if(jachutou(palavrasecreta[i], chutes, tentativas)) {
            printf("%c ", palavrasecreta[i]);
        } else {
            printf("_ ");
        }

    }
    printf("\n");

}
```

Fazendo uso dessa função no lugar certo, o loop principal do jogo ficou mais simples de ser lido e entendido. Agora, dá até para ler em português o que acontece. Ele "desenha forca" e "chuta", enquanto "não acertou" e "não enforcou". Veja que você consegue entender o algoritmo em alto nível em poucos segundos. O desenvolvedor só precisará ler os seus detalhes ou seja, a implementação de cada uma dessas funções se quiser ou tiver necessidade.

Nosso código está cada vez mais bem escrito e isolado. Isso é fundamental quando programamos.

```
do {

    desenhaforca(palavrasecreta, chutes, tentativas);
    chuta(chutes, &tentativas);

} while (!acertou && !enforcou);
```

Está perdido?

O código até o momento está em http://pastebin.com/ERgqEeNb

Por fim, ainda podemos deixar mais fácil de ser entendida a parte na qual selecionamos a palavra secreta. Por enquanto, ela é fixa, porém, mais à frente, vamos ler de um arquivo de palavras. Se já tivermos esse código bem isolado em uma função, ficará fácil depois trocar. A função `escolhepalavra()` receberá o array `palavrasecreta`:

```
void escolhepalavra(char* palavrasecreta) {
    sprintf(palavrasecreta, "MELANCIA");
}
```

Vamos invocá-la também em um lugar mais oportuno, de melhor legibilidade, um pouco mais afastado da declaração das variáveis. Logo abaixo da invocação da função `abertura()`, por exemplo, parece bom. A função `main()` agora está bastante enxuta:

```
abertura();
escolhepalavra(palavrasecreta);

do {
    // ...
} while (...);
```

Está perdido?

O código até o momento está em http://pastebin.com/fJ3JJMXB

11.9 Variáveis globais

Repare que todas nossas funções agora precisam passar os mesmos parâmetros para lá e para cá. Faz sentido; afinal, o array de chutes, a palavra secreta e

o número de tentativas são variáveis importantes para o jogo. Esse problema, inclusive, nos serviu de motivação para nossa discussão de funções, parâmetros e ponteiros.

E se conseguíssemos declarar uma variável que fosse visível para mais de uma função, e não só para aquela na qual ela foi declarada? Nosso código ficaria bem menor, uma vez que a **assinatura da função** assinatura é o nome que damos para o nome, parâmetros e tipos de retorno de uma função seria bem mais simples.

Na verdade, podemos. Em C, podemos declarar uma variável fora do arquivo. Essas variáveis são chamadas de **variáveis globais**, pois elas ficam visíveis e passíveis de serem acessadas por qualquer função declarada em nosso programa. Elas são bastante úteis quando temos variáveis que são importantes e, por consequência, manipuladas por todas as nossas funções.

Vamos fazer isso com as variáveis `chutes`, `palavrasecreta` e `tentativas`. Vamos declará-las logo após a lista de *includes*:

```
#include <stdio.h>
#include <string.h>

char palavrasecreta[20];
char chutes[26];
int tentativas = 0;
```

Já que agora elas são visíveis para todas as funções, não precisamos mais passá-las por parâmetro em todas as funções. Vamos remover os que não são mais necessários.

Na função `chuta()`, não precisamos mais nem de `chutes` nem de `tentativas` como parâmetro. Já que `tentativas` agora é um inteiro convencional (e não mais um ponteiro), vamos tirar as operações de ponteiro e fazer operações comuns com ela:

```
void chuta() {
    char chute;
    printf("Qual letra? ");
    scanf(" %c", &chute);

    chutes[tentativas] = chute;
```

```
    tentativas++;
}
```

A função `jachutou()` também não precisa mais do array de `chutes`, nem da variável `tentativas`. Aqui a mudança é simples, e basta removermos da assinatura da função. Repare que, ainda assim, precisamos receber a `letra` que procuraremos no array. Veja como ela ficou mais simples:

```
int jachutou(char letra) {
    int achou = 0;
    for(int j = 0; j < tentativas; j++) {
        if(chutes[j] == letra) {
            achou = 1;
            break;
        }
    }

    return achou;
}
```

A função `desenhaforca()` também não precisa receber mais nenhum parâmetro, nem passar todos eles para a função `jachutou()`, invocada dentro dela.

```
void desenhaforca() {

    printf("Você já deu %d chutes\n", tentativas);

    for(int i = 0; i < strlen(palavrasecreta); i++) {

        if(jachutou(palavrasecreta[i])) {
            printf("%c ", palavrasecreta[i]);
        } else {
            printf("_ ");
        }

    }
    printf("\n");

}
```

O mesmo acontece com a função `escolhepalavra()`:

```
void escolhepalavra() {
    sprintf(palavrasecreta, "MELANCIA");
}
```

Por fim, a `main` também não precisa mais passar os parâmetros para as funções. Veja só como ela ficou totalmente enxuta e fácil de ser lida. Repare no `tentativas` no fim do loop, já que não podemos mais fazer o incremento dentro da função `chuta`:

```
int main() {

    int acertou = 0;
    int enforcou = 0;

    abertura();
    escolhepalavra();

    do {

        desenhaforca();
        chuta();

        tentativas++;

    } while (!acertou && !enforcou);

}
```

Veja que agora escolhemos o melhor lugar para declarar nossas variáveis.

Variáveis que fazem sentido existirem em um contexto mais amplo e são diretamente relacionadas com o nosso jogo de forca `palavrasecreta`, `chutes` e `tentativas`, são globais e visíveis por todas as funções. Já variáveis que são mais locais e específicas de uma função em particular, como a variável `achou` na função `jachutou` e a `i` na função `desenhaforca`, são declaradas locais e existem só naquele contexto.

Será, então, que variáveis globais são sempre boas? Algo para você pensar desde o começo de sua carreira em desenvolvimento de software é que toda

decisão de código tem sempre prós e contras. Sem exceção. Ainda não fomos capazes de criar balas de pratas, ou seja, soluções perfeitas e exatas para todos os problemas. Variáveis globais nos ajudam a diminuir a quantidade de código que escrevemos, uma vez que precisamos passar menos parâmetros para lá e para cá, e as funções já as acessam diretamente.

A desvantagem é que agora fica mais difícil entender quem modificou o quê. Antes, quando as variáveis estavam sempre declaradas dentro de escopos controlados, se você quisesse saber quem mudou a variável `tentativas`, bastava olhar a função em que ela estava declarada. Agora, você precisa olhar o código todo. Ou seja, perdemos um pouco do controle sobre o acesso a elas, já que qualquer função pode alterá-las. É uma troca.

Variáveis globais são geralmente muito criticadas por isso. Mas, se você usá-las bem, não terá problemas. Lembre-se de colocar como global apenas aquelas que tenham um sentido maior para o código. Mais à frente (e no mundo real), criaremos programas com vários arquivos. Esse é o real problema: ter muitos arquivos diferentes, que fazem coisas diferentes, mexendo nas mesmas variáveis. Imagine o código do seu jogo de adivinhação mexendo nas variáveis do jogo de forca? Não faz sentido. Mesmo que inventássemos alguma regra em que os jogos se combinassem, o ideal seria que só o arquivo `forca.c` mexesse em variáveis do jogo de forca, e o `adivinhacao.c` mexesse em variáveis do jogo de adivinhação.

Neste livro, não falamos sobre Orientação a Objetos, mas sem dúvida, é o próximo assunto a respeito do qual você deve ler. **Orientação a Objetos** é uma outra maneira de escrever programas de computador. Sua maneira de pensar é diferente da **programação procedural**, sendo esta que usamos aqui com a linguagem C, onde temos um conjunto de funções que trabalham juntas.

Existem muitas maneiras diferentes de se escrever um programa de computador e, em todas elas, você deve pensar em facilitar a manutenção do seu código. Como fazer isso? Deixando sempre perto as coisas que mudam juntas. Tudo que é relacionado ao jogo da forca fica em um arquivo, e tudo que é relacionado ao de adivinhação fica em outro.

Nosso jogo agora, além de bonito por fora, está ficando também bonito por dentro.

Está perdido?

O código até o momento está em http://pastebin.com/HAC8XEkx

11.10 Resumindo

Neste capítulo, aprendemos:

- A escrever funções próprias;
- A fazer uso das funções declaradas;
- O que são ponteiros;
- Como usar ponteiros em C;
- Que arrays são ponteiros;
- A declarar funções com parâmetros;
- A criar funções com retorno;
- A usar variáveis globais;
- Quando não usar variáveis globais.

Capítulo 12

Entrada e saída (I/O)

Falta pouco para terminarmos nosso jogo. Nesse momento, precisamos fazer toda a lógica de vencer ou perder o jogo. Em um jogo de forca, perdemos quando esgotamos todas nossas chances, e ganhamos quando acertamos a palavra secreta. Mas, agora que já aprendemos a criar funções, tudo fica mais fácil. Antes, precisávamos olhar aquelas centenas de linhas de código e achar o ponto exato. Agora, podemos escrever o código em funções isoladas, e depois apenas invocá-las.

Vamos começar escrevendo a função que nos diz que o jogador perdeu o jogo. Basta contarmos o número de chutes incorretos que ele deu. Caso este seja maior que o número de tentativas permitidas, ele perdeu. Nossa função denominada `enforcou()` nos devolverá 1, se ele foi enforcado; ou 0, se ainda não foi. O número de tentativas será 5 (cabeça, corpo, braços, pernas e fim).

A lógica dessa função será a seguinte: declararemos inicialmente uma variável chamada `erros`, que contará o número de letras que o usuário chutou

de maneira incorreta. Essa variável nos será útil ao final, pois, se ela for maior ou igual a 5, o jogador deverá ser enforcado. Faremos um loop na lista de chutes e, para cada chute dado, verificaremos se a letra existe na palavra secreta. Se existir, quebramos o loop e passamos para a frente. Aqui, utilizaremos a mesma estratégia que estamos usando quando temos loops encadeados. A variável `existe` nos ajudará a saber se a letra foi encontrada no loop de dentro. Caso contrário, somaremos 1 à variável `erros`, e continuamos.

Veja, em código:

```
int enforcou() {

    int erros = 0;

    // vamos fazer o loop em todos os chutes dados
    for(int i = 0; i < tentativas; i++) {

        int existe = 0;

        // agora vamos olhar letra a letra da palavra secreta
        // e ver se encontramos o chute aqui
        for(int j = 0; j < strlen(palavrasecreta); j++) {
            if(chutes[i] == palavrasecreta[j]) {

                // encontramos, vamos quebrar o loop
                existe = 1;
                break;
            }
        }

        // se não encontrou, soma um na quantidade de erros
        if(!existe) erros++;
    }

    // se tivermos mais do que 5 erros, retornamos 1
    // caso contrário, retornamos 0.
    return erros >= 5;
}
```

Repare no `return` em uma linha. Acostume-se com códigos assim.

Lembre-se de que a máquina vai primeiro avaliar o elemento da direita ou seja, a expressão condicional , para depois fazer o retorno. Em outras palavras, se `erros` for maior que 5, ela avaliará isso para 1 (verdadeiro); caso contrário, 0. E a função retornará o resultado disso.

Precisamos agora invocar essa função. Como ela está bem isolada, podemos facilmente trocar a variável `enforcou`, que havíamos declarado lá na nossa `main` (mas que nunca usamos), pela invocação da função. Precisamos, claro, deletar a variável:

```
int main() {

    int acertou = 0;

    abertura();
    escolhepalavra();

    do {

        desenhaforca();
        chuta();

    // aqui invocamos a função enforcou
    } while (!acertou && !enforcou());

}
```

Se você rodar o programa agora e fizer 5 tentativas erradas, verá que nosso programa parará.

Está perdido?

O código até o momento encontra-se em http://pastebin.com/KndnEWkj

Conforme vamos desenvolvendo nosso software e aprendendo com ele, mudamos de ideia. Por exemplo, a variável `tentativas`, cujo objetivo era contar a quantidade de chutes dados, agora ficou ambígua. Com esse nome,

também poderia ser o número de chances que ele tem. Se um novato pegar esse código, ele precisará ler mais do código para entender.

Vamos, portanto, **refatorar** nosso código. Refatorar é mudar o código, geralmente para melhor, mas sem mudar o que ele faz. Ou seja, deixá-lo mais bonito e fácil de ser mantido. Uma refatoração comum é, por exemplo, mudar o nome das coisas, como variáveis, métodos ou até mesmo arquivos.

Trocaremos o termo `tentativas` por `chutesdados`, que deixa bem mais claro o que ela guarda lá. Precisamos trocar na declaração e, óbvio, em todos os lugares que a usamos:

```
int chutesdados = 0;
```

A próxima etapa é descobrir se ganhamos ou não o jogo. A abordagem será a mesma, porém *ao contrário*. Passaremos primeiro pela palavra secreta. Para cada letra, descobriremos se o usuário já a chutou. Caso alguma letra não tenha sido chutada, podemos matar logo a função e retornar falso. Ao final, se ele chutou todas as letras que existem na palavra, ele ganhou o jogo.

Essa função é bem mais simples, afinal já temos a função `jachutou` que nos diz se uma letra já foi chutada ou não. Vamos apenas passear pela palavra secreta e ver se a letra já foi chutada. Se ela não foi, então negaremos o `if` e faremos um `return 0`. Note que podemos ter `return`s a qualquer momento do nosso código; a máquina parará a função naquele momento e retornará o resultado direto. Vamos declarar essa função logo abaixo da `jachutou()`:

```
int ganhou() {
    for(int i = 0; i < strlen(palavrasecreta); i++) {
        if(!jachutou(palavrasecreta[i])) {
            return 0;
        }
    }

    return 1;
}
```

Para vê-la funcionar, basta também colocarmos sua invocação dentro do `do-while` e remover a inútil variável `acertou`. Pronto, nosso jogo agora para se o usuário ganhou ou perdeu:

```
int main() {

    abertura();
    escolhepalavra();

    do {

        desenhaforca();
        chuta();

    } while (!ganhou() && !enforcou());

}
```

Nosso jogo agora tem começo, meio e fim. Só não está bonito, mas o embelezamos mais tarde.

Erro de compilação?

Seu código não está compilando? Pode acontecer. Leia a próxima seção, na qual resolveremos o seu possível problema.

Está perdido?

O código até o momento está em http://pastebin.com/7eg47h7N

12.1 Header files

É provável que, nesse momento, o código de alguns leitores deste livro compile, e de outros não. Repare que em nenhum momento comentei onde devemos colocar cada função que declaramos. Será que existe uma ordem específica? Sim, existe.

O compilador da linguagem C não é tão esperto, e a ordem da declaração das funções é importante. Quando o compilador está passando pelo arquivo,

se ele encontra uma invocação de função que não conhece, ele opta por reclamar, em vez de continuar e procurar pela declaração mais à frente (isso é feito por compiladores de linguagens mais modernas). O erro que você deve ter tomado é parecido com o seguinte:

```
forca.c:31:7: warning: implicit declaration of function
'jachutou' is invalid in C99 [-Wimplicit-function-declaration]
```

Entretanto, existe uma maneira de resolver isso: declarando todas as assinaturas de funções que aparecerão naquele arquivo, antes de darmos o código delas. A assinatura de uma função é aquela primeira linha que usamos para declará-la, que contém o tipo de retorno, o nome e os parâmetros que ela recebe.

Vamos colocá-las todas, uma a uma, logo após os *includes*. Aqui optamos por colocar antes das variáveis globais, mas não há diferença.

```c
#include <stdio.h>
#include <string.h>

// lista de funções que aparecerão no arquivo
int enforcou();
void abertura();
void chuta();
int jachutou(char letra);
int ganhou();
void desenhaforca();
void escolhepalavra();

// variáveis globais
char palavrasecreta[20];
char chutes[26];
int chutesdados = 0;

// agora aqui começam as implementações das funções,
// do jeito que você está acostumado.
```

Note que agora você pode ter as suas funções em qualquer ordem. Coloque a função `ganhou()` antes da `jachutou()`, por exemplo, e veja que o código compila normalmente.

Esse é um problema comum de programas escritos em C, e ele é ainda mais grave quando temos nosso código espalhado em arquivos diferentes e queremos usar funções de um arquivo em outro. Por esse motivo, apesar de nosso código funcionar com a declaração dessas funções, temos o costume de colocar isso em um arquivo separado. Esse arquivo é conhecido como **header file** e tem a extensão `.h`.

Repare que você já viu essa extensão antes. É a mesma extensão do `stdlib`, `string` e todos os outros que importamos. Ou seja, em algum lugar, existe um `stdlib.c` que contém a implementação das funções. Como queremos usá-las em nosso programa, importamos suas declarações, que estão todas no `.h`.

Vamos criar o arquivo `forca.h` e colocar essas declarações lá:

```
// lista de funções que aparecerão no arquivo
int enforcou();
void abertura();
void chuta();
int jachutou(char letra);
int ganhou();
void desenhaforca();
void escolhepalavra();
```

Agora, em nosso `forca.c`, o importaremos. Repare que aqui usamos aspas em vez dos sinais de maior e menor. Afinal, esse é um header file que é nosso e está no diretório corrente:

```
#include <stdio.h>
#include <string.h>
#include "forca.h"

// variáveis globais
char palavrasecreta[20];
char chutes[26];
int chutesdados = 0;

// agora aqui começam as implementações das funções,
// do jeito que você está acostumado.
```

Agora, sempre que criarmos uma nova função, precisamos declará-la também no nosso header file. Acostume-se com isso, pois, em nosso próximo jogo, dividiremos melhor os arquivos e usaremos funções para lá e para cá, e eles serão importantes.

12.2 Lendo arquivos

A única parte chata do jogo é que a palavra secreta está fixa. Depois que ele adivinhar a primeira vez, acabou. Precisamos dar um jeito de fazer com que a palavra seja randômica. Criar números randômicos é fácil, mas criar palavras que façam sentido é, com certeza, um problema computacional bastante desafiador.

O que faremos é escolher uma palavra aleatória de um arquivo de texto, que será nosso "banco de dados de palavras". Esse arquivo pode ter qualquer tamanho, ou seja, você poderá colocar novas palavras lá sempre que quiser, e o programa deve entender isso.

Vamos começar escrevendo um arquivo de exemplo. Sempre que quisermos criar um arquivo que será lido pelo computador, precisamos definir um formato. Como separaremos as palavras? Por vírgula? Por *enter*? O formato é importante, afinal, o programa precisará saber como interpretá-lo.

Nosso arquivo terá o seguinte formato: a primeira linha nos indicará quantas palavras existem naquele arquivo. Em seguida, teremos várias palavras, uma em cada linha. Isso facilitará a nossa leitura via código. Veja o arquivo, que chamaremos de `palavras.txt`:

```
3
MELANCIA
MORANGO
MELAO
```

Agora vamos começar a lê-lo. A primeira coisa que precisamos fazer é abri-lo. Para isso, precisamos declarar um ponteiro do tipo `FILE*`, e usar a função `fopen()`, que abre um arquivo do disco. Essa função nos devolve, então, um ponteiro que guardaremos na variável já declarada. A partir daí, passamos esses ponteiros para as funções que lerão os caracteres desse ar-

quivo. Repare no "r". Isso indica que estamos abrindo-o somente para leitura; entretanto, também poderíamos tê-lo aberto para escrita.

Todo o código estará dentro da função escolhepalavra():

```
void escolhepalavra() {
    FILE* f;

    f = fopen("palavras.txt", "r");
}
```

Essa é a primeira vez que estamos lidando com recursos de I/O, ou seja, de entrada ou saída. Na prática, elas podem falhar, pois o arquivo pode não estar disponível, ou não termos permissão de leitura etc. Sempre que lidamos com isso, precisamos tratar as possíveis falhas.

A função fopen() devolve um ponteiro com endereço 0, se algum erro ocorrer. Vamos tratar esse possível erro. Caso ele tenha ocorrido, devolveremos uma mensagem para o usuário e acabaremos com nosso programa ali mesmo, usando a função exit() que finaliza a aplicação. Repare que o parâmetro 1, passado ao exit, indica ao sistema operacional que o programa terminou de maneira que ele não gostaria; ou seja, um erro ocorreu. Se tivéssemos retornado 0, o sistema operacional saberia que o programa rodou de acordo com o esperado. A função exit() está declarada em stdlib.h. Precisamos colocar o include:

```
#include <stdlib.h>

void escolhepalavra() {
    FILE* f;

    f = fopen("palavras.txt", "r");
    if(f == 0) {
        printf("Banco de dados de palavras não disponível\n\n");
        exit(1);
    }
}
```

Agora sim, estamos prontos para começar a ler nosso arquivo, e sabemos que na primeira linha temos a quantidade de palavras que leremos na sequência. Vamos capturar esse número. Você já sabe mais ou menos como ler do

arquivo. Em C, temos a função `fscanf()`, similar à que você já conhece. Sua diferença é que o primeiro parâmetro é o ponteiro para um arquivo.

Se quisermos ler um inteiro de um arquivo, por exemplo, fazemos igual ao código a seguir. Repare que a função sabe que o número "acabou" quando encontra um *enter* no arquivo. Ou seja, nesse caso, ela lê até achar um fim de linha.

```
int qtddepalavras;
fscanf(f, "%d", &qtddepalavras);
```

Toda vez que lemos de um arquivo, a leitura é sequencial. Em outras palavras, se já capturamos o primeiro número, a próxima vez que chamarmos a função `fscanf()`, ela continuará a leitura do ponto que parou. É como se ela tivesse uma pequena seta que aponta para a posição que está lendo no momento.

Vamos ler, então, a próxima coisa que aparece no arquivo. Sabemos que é uma palavra. Vamos declarar um array e ler, usando a máscara `%s`. Dado que em nosso exemplo sabemos que temos 3 palavras, vamos ler 3 palavras de uma só vez, apenas para teste:

```
char palavra1[50];
fscanf(f, "%s", palavra1);

char palavra2[50];
fscanf(f, "%s", palavra2);

char palavra3[50];
fscanf(f, "%s", palavra3);
```

Nesse momento, se imprimirmos as 3 strings, teremos `MELANCIA`, `MORANGO` e `MELAO`, igual esperávamos. Porém, nesse nosso jogo em particular, não precisamos salvar todas elas, mas sim somente uma (a que escolhermos de maneira aleatória). Vamos, portanto, gerar um número aleatório, de 0 até a quantidade de elementos, e salvar apenas a palavra que estiver naquela posição.

Não temos como pular direto para a linha que queremos, então precisaremos fazer um loop com a função `fscanf()` e, na linha certa, salvar o

retorno. Para facilitar, vamos salvar a palavra secreta já na variável correta `palavrasecreta`. Repare que a última interação do loop será justamente a linha randômica escolhida. O loop acabará e a variável terá a palavra secreta que queremos:

```
int qtddepalavras;
fscanf(f, "%d", &qtddepalavras);

// gera número aleatório
// não esqueça de incluir time.h
srand(time(0));
int randomico = rand() % qtddepalavras;

// lê do arquivo até chegar na linha desejada
for(int i = 0; i <= randomico; i++) {
    fscanf(f, "%s", palavrasecreta);
}
```

Nosso código está quase pronto. Já selecionamos uma palavra aleatória dentro do nosso banco de dados de palavras. Só estamos nos esquecendo de uma coisa: sempre que lidamos com I/O, precisamos lembrar de abrir o arquivo ou qualquer outra fonte de entrada , usá-lo e, ao final, fechá-lo.

Fechar é importante. Ao fazer isso, aquele recurso é liberado para que outros programas consigam fazer uso dele também. Quando abrimos um arquivo para leitura, o sistema operacional pode bloquear sua leitura em outros aplicativos, até que o programa corrente termine seu trabalho. Ou seja, feche o mais rápido possível. Fazemos isso com a função `fclose()`.

O método `escolhepalavra()` completo ficou:

```
void escolhepalavra() {
    FILE* f;

    f = fopen("palavras.txt", "r");
    if(f == 0) {
        printf("Banco de dados de palavras não disponível\n\n");
        exit(1);
    }
```

```
    int qtddepalavras;
    fscanf(f, "%d", &qtddepalavras);

    srand(time(0));
    int randomico = rand() % qtddepalavras;

    for(int i = 0; i <= randomico; i++) {
        fscanf(f, "%s", palavrasecreta);
    }

    fclose(f);
}
```

Repare que, ao longo desta seção, só alteramos um ponto do nosso arquivo, que foi a função `escolhepalavra()`. E tudo funciona. Veja só como dividir nosso programa em funções bem definidas é realmente útil. Podemos fazer mudanças em pequenas partes do programa sem nos preocuparmos com o todo. É isso que faz um código ser fácil de ser mantido: quando conseguimos mexer apenas em pontos específicos, sem a necessidade de ler ou alterar todo o código.

Está perdido?

O código até esse momento está em http://pastebin.com/cAVVZFVw

12.3 Escrevendo no arquivo

A próxima funcionalidade em nosso jogo é perguntar ao usuário se ele deseja adicionar uma nova palavra no banco de dados. Como ela funcionará? Ao final do jogo, perguntaremos se ele deseja fazer isso. Se ele responder `S` (sim), perguntaremos a palavra e a adicionaremos ao arquivo.

Vamos começar pela parte fácil, que é perguntar se o usuário quer ou não adicionar uma palavra. E isso, claro, em uma nova função, isolada do resto do nosso programa. Lembre-se de que, como vamos capturar um único *char*,

precisamos colocar um espaço na máscara, para que o `scanf()` ignore o *enter* ao final:

```
void adicionapalavra() {
    char quer;

    printf("Você deseja adicionar uma nova palavra no
            jogo (S/N)?");
    scanf(" %c", &quer);

}
```

Se ele responder `S`, perguntaremos a ele a palavra. Também não há novidade:

```
void adicionapalavra() {
    char quer;

    printf("Você deseja adicionar uma nova palavra no
            jogo (S/N)?");
    scanf(" %c", &quer);

    if(quer == 'S') {
        char novapalavra[20];

        printf("Digite a nova palavra, em letras maiúsculas: ");
        scanf("%s", novapalavra);

        // agora falta salvar no arquivo
    }

}
```

Agora precisamos salvar no nosso arquivo de banco de dados. Para abrir um arquivo, precisamos fazer uso da função `fopen()`, que já conhecemos. O que mudará são os parâmetros que passaremos a ela. O `r` indicava leitura. Usaremos aqui o `a`, que significa *append*, ou seja, que vamos inserir no fim do arquivo.

Com um `FILE` aberto, tudo o que precisamos fazer é usar a função `fprintf()`, análoga à `printf()`, que você já conhece. A diferença é que o primeiro parâmetro que ela recebe é justamente o arquivo em que ela escreverá. Você percebeu que o `f` tanto da `fprintf` quanto da `fscanf` é de *file*?.

Também não esqueceremos de tratar um possível erro na abertura do arquivo. Ele pode não ter acontecido no começo do jogo, mas pode acontecer ao final. Em código:

```
if(quer == 'S') {
    char novapalavra[20];

    printf("Digite a nova palavra, em letras maiúsculas: ");
    scanf("%s", novapalavra);

    FILE* f;

    // abre arquivo
    f = fopen("palavras.txt", "a");
    if(f == 0) {
        printf("Banco de dados de palavras não disponível\n\n");
        exit(1);
    }

    // escreve a palavra nele
    fprintf(f, "%s", novapalavra);

    // fecha
    fclose(f);

}
```

Até o momento, aprendemos que o ponteiro `FILE*` aponta para um arquivo, e passamos esse ponteiro para todas as outras funções que o manipulam. O `fscanf()` consegue ler dados de um arquivo e o `fprintf` consegue escrever nele. Mas tudo depende da configuração passada para a função `fopen()`. Se quisermos apenas ler, usamos `"r"`, e se quisermos *appendar* ou escrever ao final, usamos `"a"`.

Não esqueça de fechar o arquivo

Fechar o arquivo com o `fclose()` é especialmente importante em momentos de escrita. Isso porque muitas vezes a máquina opta por não escrever no disco no momento em que você invoca um `fprintf()`; ela prefere "cachear", ou seja, guardar a modificação em memória e só salvar no disco depois. Ela toma essa decisão porque ficar indo ao disco e escrevendo fisicamente nele, de pouco em pouco, pode ser uma operação demorada. Ela prefere esperar mais texto e ir de uma só vez ao disco rígido.

A operação de fechar, portanto, diz à máquina que você acabou tudo o que tinha que escrever e que agora, independente do tamanho desse *buffer*, ela precisa escrever no disco. Se o programador, por descuido, esquece de fechar o arquivo, a máquina não escreverá nele, e ele terá a sensação de que seu código não funcionou. Assim, feche sempre o arquivo que você abriu.

Mas veja que somente escrever ao final não é suficiente. Nosso arquivo de banco de dados tem um formato diferente, pois a primeira linha indica a quantidade de palavras. Ou seja, além de adicionar a palavra, precisamos ainda modificar o número que está no começo. Para isso, precisamos fazer uso de um outro modo de abertura de arquivo. O `"w"`, por exemplo, nos ajuda a escrever em arquivos, mas ele só escreve em arquivos novos. Em outras palavras, se o arquivo existir, ele o apaga primeiro.

O modo que precisamos aqui é o `r+`. Ele nos permite ler e alterar um arquivo, sendo exatamente o que precisamos. O primeiro passo é ler o número que está no começo do arquivo, incrementá-lo e sobrescrevê-lo. Em seguida, adicionar a nova palavra ao final.

Vamos começar abrindo o arquivo nesse novo modo e lendo o número que está na primeira linha:

```
f = fopen("palavras.txt", "r+");
if(f == 0) {
    printf("Banco de dados de palavras não disponível\n\n");
```

```
    exit(1);
}

int qtd;
fscanf(f, "%d", &qtd);
```

Agora, precisamos incrementar essa variável e sobrescrevê-la no arquivo. Porém, como já lemos o inteiro, o "ponteiro" que aponta para o lugar do arquivo que a leitura continuará está mais à frente. Precisamos voltar esse ponteiro e posicioná-lo no lugar que desejamos começar a escrever. Para mudar o ponteiro de posição, usamos a função `fseek()`. Ela recebe três parâmetros: o arquivo, quantos bytes ela deve andar para a esquerda ou para a direita e de onde ela deve começar a andar do começo do arquivo, da posição corrente ou do final. Vamos posicioná-la "0 bytes" a partir do começo do arquivo:

```
qtd++;

fseek(f, 0, SEEK_SET);
```

Repare que `SEEK_SET` é uma constante. Poderíamos usar também a `SEEK_CUR` para andar a partir do ponto atual; ou mesmo `SEEK_END`, para andar a partir do fim do arquivo. Com o ponteiro posicionado na posição que queremos, basta agora escrevermos a variável `qtd`, usando o `fprintf()`. A máquina não pensará duas vezes e escreverá por cima do que está naquela posição:

```
fprintf(f, "%d", qtd);
```

Agora, por fim, precisamos escrever a palavra no final do arquivo. Vamos novamente usar `fseek()` para ir ao fim do arquivo, e `fprintf()` para escrever a palavra. Repare no `\n` no começo da string. Precisamos disso, pois o ponteiro nos posiciona bem ao final do arquivo que, teoricamente, não tem um *enter* no fim. Escrevemos um *enter* e, aí sim, a palavra:

```
fseek(f, 0, SEEK_END);
fprintf(f, "\n%s", novapalavra);
```

Pronto. Agora o jogador pode adicionar novas palavras ao final da partida:

```
void adicionapalavra() {
    char quer;

    printf("Você deseja adicionar uma nova palavra no
            jogo (S/N)?");
    scanf(" %c", &quer);

    if(quer == 'S') {
        char novapalavra[20];

        printf("Digite a nova palavra, em letras maiúsculas: ");
        scanf("%s", novapalavra);

        FILE* f;

        f = fopen("palavras.txt", "r+");
        if(f == 0) {
            printf("Banco de dados de palavras não
                    disponível\n\n");
            exit(1);
        }

        int qtd;
        fscanf(f, "%d", &qtd);
        qtd++;
        fseek(f, 0, SEEK_SET);
        fprintf(f, "%d", qtd);

        fseek(f, 0, SEEK_END);
        fprintf(f, "\n%s", novapalavra);

        fclose(f);

    }

}
```

Está perdido?

O código até o momento está em http://pastebin.com/FAM8pA1w

12.4 Mais sobre I/O

A biblioteca de entrada e saída da linguagem C é bastante extensa. Mesmo as funções estudadas aqui podem fazer muito mais do que o mostrado. Todas as funções, por exemplo, possuem retorno. A própria `fscanf()` retorna o número de itens devolvidos. Por exemplo, se a máscara for `%d`, ela retornará 1, caso tudo dê certo. Se a máscara for `%d %d`, ou seja, capturar 2 inteiros, ela retornará 2.

```
int n1, n2;

int sucesso = fscanf("%d %d", &n1, &n2);
if(sucesso == 2) {
    printf("Os dois números: %d e %d", n1, n2);
}
else {
    printf("Aconteceu um erro");
}
```

Podemos verificar também se o ponteiro atual do arquivo está apontando para o fim. Para isso, basta usar a função `feof()`. Ele pode ser bastante útil quando queremos ler o arquivo inteiro até o fim. Por exemplo, se quisermos ler caractere por caractere do arquivo e imprimi-lo, precisamos repetir isso enquanto não chegarmos ao fim do arquivo.

Por enquanto, a única função de leitura que vimos até então é a `fscanf()`. Ela é importante quando queremos ler caracteres ASCII e strings. Mas, muitas vezes, precisamos ler um byte, um char, ou mesmo uma *struct* (que aprenderemos no próximo jogo). Nesses casos, precisamos ler e escrever dados de maneira "mais bruta". A função `fgetc()`, por exemplo, lê apenas o próximo `char`.

```
FILE* f;
char c;

f = fopen("arquivo.txt", "r");
while(!feof(f)) {
    c = fgetc(f);
    printf("Char %c\n", c);
}
```

Nesses casos, também as funções `fread()` e `fwrite()` podem ajudar. Elas leem uma quantidade específica de bytes e as guardam na estrutura que você passar, seja ela um array, uma variável ou uma struct. No próximo jogo, faremos uso delas.

Perceba que daqui para frente você começará a aprender e fazer uso de funções que são complexas e fazem muita coisa. A melhor maneira de entendê-las melhor é ler o manual oficial. Nisso, o Google o ajudará. Sempre que tiver dúvida dos parâmetros que uma função recebe, o que ela retorna e como interpretar esse retorno, procure pela sua definição na internet. Buscar é também parte do dia a dia do programador.

12.5 Resumindo

Neste capítulo, aprendemos:

- A criar e usar header files próprios;
- A manipular entrada e saída;
- A abrir arquivos com `fopen`;
- A usar modos diferentes de abrir arquivo, como só leitura ou só escrita;
- A ler e escrever com `fscanf` e `fprintf`;
- Que a biblioteca de I/O é grande, e precisamos buscar sempre pelo manual das funções.

Capítulo 13

Finalizando o jogo

Nosso jogo está inteiro funcional. Falta agora o deixarmos mais apresentável para o jogador. Afinal, interface é importante. Vamos melhorá-la, e aproveitar para aperfeiçoar alguns detalhes também no código, pois já sabemos que, quanto mais legível, melhor.

13.1 Evitando repetição de código

Repetição de código é algo sempre problemático. Se temos o mesmo pedaço de código em muitos pontos da aplicação, isso implica em alterar todos os pontos quando necessário. Infelizmente, na prática, muitas vezes não propagamos a alteração para todos os lugares necessários, e isso nos traz bugs.

Você já tem ferramentas para evitar esse tipo de problema. Se há um trecho de código que é comum para várias partes do seu programa, basta extraí-lo e colocá-lo em uma função. Funções são facilmente reutilizáveis. E se você

alterá-las, todos os lugares que as invocam farão uso da nova função.

Também já vimos que quando temos algum "número mágico" ou seja, um número fixo , podemos colocá-lo em uma constante. A constante pode ser facilmente reaproveitada e possui um nome semântico, muito melhor do que deixar o número jogado. E, se a alterarmos, a mudança propagará para todos os lugares que a usam.

Nosso jogo de forca tem um número mágico, que já está espalhado pelo código. Veja que o tamanho máximo de uma palavra em nosso jogo é 20 caracteres. Isso está definido na declaração do array `palavrasecreta`. Se quisermos mudar o tamanho da palavra, precisaremos mudar a declaração do array. Mas não é só isso: é necessário mudar também dentro da função `adicionapalavra()`, pois lá declaramos um array com o mesmo tamanho. No momento, são apenas 2 pontos; porém amanhã serão 3, e depois de amanhã, 4.

Vamos já corrigir isso, criando uma constante para isso. E, dessa vez, vamos colocá-la no header file. Faremos um `#define` por lá, e o reutilizaremos em nosso código. Veja:

```
// forca.h
#define TAMANHO_PALAVRA 20

// forca.c
char palavrasecreta[TAMANHO_PALAVRA];

void adicionapalavra() {
    // ...

    char novapalavra[TAMANHO_PALAVRA];
}
```

13.2 Extraindo funções

Até então, só mostramos o número de chutes dados pelo jogador, mas a parte mais legal do jogo de forca é justamente exibir o homenzinho enforcado. Vamos lá, então, fazer um desenho mais amigável dentro da função `desenhaforca()`:

```
void desenhaforca() {

    printf("  _______       \n");
    printf(" |/      |      \n");
    printf(" |      (_)     \n");
    printf(" |      \\|/     \n");
    printf(" |       |      \n");
    printf(" |      / \\     \n");
    printf(" |              \n");
    printf("_|___           \n");
    printf("\n\n");

    // ...
}
```

Entretanto, não podemos exibir o boneco cheio o tempo inteiro. Isso depende do número de erros. Se o número de erros é 0, exibimos apenas a corda. Se ele errou 1 vez, exibimos a cabeça; 2 vezes, a cabeça e o corpo; 3 vezes, a cabeça, o corpo e os braços; 4 vezes, a cabeça, o corpo, os braços e as pernas.

Precisamos saber o número de vezes que o jogador já errou. Já temos algo parecido na função `enforcou()`. Ela calcula o número de erros e depois nos retorna se essa quantidade é maior que 5. Mas ela não nos serve exatamente, porque precisamos saber a quantidade exata de erros. Ou seja, precisamos só de um pedaço dessa função.

Obviamente, não temos como utilizar parte de uma função. Porém, podemos quebrá-la em duas outras menores. Vamos dividir essa função `enforcou()` em duas: uma, nova, `chuteserrados()`, que calcula o número de chutes errados; e a outra, já existente, que fará uso da nova função e nos retornará verdadeiro se esse número for maior que 5:

```
// extraímos para cá o pedaço daquela função
// que contava a quantidade de erros
int chuteserrados() {
    int erros = 0;

    for(int i = 0; i < chutesdados; i++) {

        int existe = 0;
```

```
        for(int j = 0; j < strlen(palavrasecreta); j++) {
            if(chutes[i] == palavrasecreta[j]) {
                existe = 1;
                break;
            }
        }

        if(!existe) erros++;
    }

    return erros;
}

int enforcou() {
    // usamos a função que acabamos de criar
    return chuteserrados() >= 5;
}
```

Quebrar funções em funções menores é parte do nosso dia a dia. Fazemos isso para reutilizar pedaços de funções já existentes, ou mesmo para aumentar a legibilidade. Veja que a função `enforcou()` agora tem uma linha de código e é bem fácil de entender o que ela faz: se ele errou mais de 5 chutes, retorna falso. Qualquer desenvolvedor gasta muito pouco tempo para entendê-la.

Vamos aproveitar e extrair mais uma função. Repare que, dentro da função `chuteserrados()`, temos um loop que nos diz se uma letra existe ou não na palavra secreta. Para a função como um todo, isso é usado para contabilizar a quantidade de chutes errados. Mas só o algoritmo que nos diz se uma letra existe na palavra nos será útil mais para frente. Extraindo, temos:

```
int letraexiste(char letra) {

    for(int j = 0; j < strlen(palavrasecreta); j++) {
        if(letra == palavrasecreta[j]) {
            return 1;
        }
    }
```

```
    return 0;
}

int chuteserrados() {
    int erros = 0;

    for(int i = 0; i < chutesdados; i++) {

        if(!letraexiste(chutes[i])) {
            erros++;
        }
    }

    return erros;
}
```

Lembre-se de que, quanto menor a função, mais fácil ela ser uma entidade e reutilizada.

13.3 Ifs ternários

Com essa função em mãos, basta agora fazemos uma sequência de `ifs`, imprimindo o boneco de acordo com o número de erros. Por exemplo, os `ifs` para 0 e 1 erros:

```
int erros = chuteserrados();

if(erros == 0) {
    printf("  _______       \n");
    printf(" |/      |      \n");
    printf(" |              \n");
    printf(" |              \n");
    printf(" |              \n");
    printf(" |              \n");
    printf(" |              \n");
    printf("_|___           \n");
}
if(erros == 1) {
```

```
    printf("  _______       \n");
    printf(" |/      |      \n");
    printf(" |      (_)     \n");
    printf(" |              \n");
    printf(" |              \n");
    printf(" |              \n");
    printf(" |              \n");
    printf("_|___           \n");
}

// e assim por diante...
```

A ideia não é ruim. Mas nosso código ficará bem extenso. Perceba que, no fim, o que queremos fazer é imprimir ou não um determinado caractere, dependendo do valor da variável `erros`. Vamos isolar esses caracteres no próprio `printf()`, fazendo uso da máscara `%c`:

```
printf("  _______       \n");
printf(" |/      |      \n");
printf(" |      %c%c%c  \n", '(', '_', ')');
printf(" |      %c%c%c  \n", '\\', '|', '/');
printf(" |       %c     \n", '|');
printf(" |      %c %c   \n", '/', '\\');
printf(" |              \n");
printf("_|___           \n");

printf("\n\n");
```

Agora, os caracteres do lado direito são os que precisamos imprimir de acordo com a variável `erros`. Veja o primeiro que aparece, o `'('`, por exemplo. Ele faz parte da cabeça; portanto, precisamos imprimi-lo se `erros >= 1`. Caso contrário, podemos imprimir "vazio".

Note que temos um `if` bem curto: se `erros >=1`, imprime `")"`; se não `" "` (vazio). Esse tipo de `if` que tem uma condição, um valor de retorno no caso verdadeiro, e outro no caso de falso, é possível de ser feito em uma só linha. Chamamos isso de **if ternário**.

A sintaxe de um if ternário é a seguinte: `(condicao ? valor verdadeiro : valor falso)`. Repare no ponto de interrogação que se-

para a condição do valor verdadeiro, e no dois-pontos que divide o valor usado no caso da condição falsa. Esse tipo de if é usado *inline*, ou seja, na mesma linha de outra instrução. Vamos usar junto com o `printf()`, por exemplo. Observe o código a seguir, quebrado em várias linhas para facilitar a legibilidade:

```
printf(" |      %c%c%c  \n",
    (erros >=1 ? '(' : ' '), // Aqui o if ternário
    '_',
    ')'
);
```

A máquina fará esse `if` na hora de executar o `printf` e, dependendo da avaliação da condição, ele retornará um valor ou outro para ser impresso. Podemos fazer isso para todas as partes do nosso boneco, cada um de acordo com a sua respectiva quantidade de erros. Olhe (e digite) esse código com carinho, pois ele é extenso e cheio de caracteres:

```
int erros = chuteserrados();

printf("  _______       \n");
printf(" |/      |      \n");
printf(" |      %c%c%c  \n", (erros>=1?'(':' '),
    (erros>=1?'_':' '), (erros>=1?')':' '));
printf(" |      %c%c%c  \n", (erros>=3?'\\':' '),
    (erros>=2?'|':' '), (erros>=3?'/': ' '));
printf(" |       %c     \n", (erros>=2?'|':' '));
printf(" |      %c %c   \n", (erros>=4?'/':' '),
    (erros>=4?'\\':' '));
printf(" |              \n");
printf("_|___           \n");
printf("\n\n");
```

Pronto. Veja como o uso do if ternário nos ajudou a economizar boas linhas de código. Imagine repetirmos esse boneco um monte de vezes? Nosso arquivo ficaria gigante.

Está perdido?

O código feito até o momento está em http://pastebin.com/z9ZDU7Bf

13.4 Últimos detalhes

Vamos agora exibir uma mensagem cada vez que ele chutar uma letra, dizendo se ele acertou ou não. Colocaremos esse código dentro da função `chute()`, a qual pega o chute do jogador. Saber se uma letra existe ou não na palavra secreta é fácil: basta invocar a função que extraímos há pouco tempo: `letraexiste()`.

```
void chuta() {
    char chute;
    printf("Qual letra? ");
    scanf(" %c", &chute);

    if(letraexiste(chute)) {
        printf("Você acertou: a palavra tem a letra %c\n\n",
            chute);
    } else {
        printf("\nVocê errou: a palavra NÃO tem a letra %c\n\n",
            chute);
    }

    chutes[chutesdados] = chute;
    chutesdados++;
}
```

Agora precisamos exibir a mensagem de ganhou ou perdeu para o usuário; afinal, o jogo está acabando sem dizer o que aconteceu. Toda essa lógica deve existir logo após o `do-while` principal. Ali, basta invocarmos novamente a função `ganhou()` e imprimir a mensagem de sucesso que queremos. Vamos aproveitar e exibir uma caveira caso ele tenha perdido:

```
if(ganhou()) {
    printf("\nParabéns, você ganhou!\n\n");

    printf("       ___________      \n");
    printf("      '._==_==_=_.'     \n");
    printf("      .-\\:      /-.    \n");
    printf("     | (|:.     |) |    \n");
    printf("      '-|:.     |-'     \n");
    printf("        \\::.    /      \n");
    printf("         '::. .'        \n");
    printf("           ) (          \n");
    printf("         _.' '._        \n");
    printf("        '-------'       \n\n");

} else {
    printf("\nPuxa, você foi enforcado!\n");
    printf("A palavra era **%s**\n\n", palavrasecreta);

    printf("    _______________         \n");
    printf("   /               \\       \n");
    printf("  /                 \\      \n");
    printf("//                   \\/\\  \n");
    printf("\\|   XXXX     XXXX   | /   \n");
    printf(" |   XXXX     XXXX   |/     \n");
    printf(" |   XXX       XXX   |      \n");
    printf(" |                   |      \n");
    printf(" \\__      XXX      __/     \n");
    printf("   |\\     XXX     /|       \n");
    printf("   | |           | |        \n");
    printf("   | I I I I I I I |        \n");
    printf("   |  I I I I I I  |        \n");
    printf("   \\_             _/       \n");
    printf("     \\_         _/         \n");
    printf("       \\_______/           \n");
}
```

Pronto. Nosso jogo está terminado. E veja o quanto você precisou aprender para fazê-lo.

Código final

O código final do jogo pode ser visto aqui:
http://pastebin.com/Qu3dDdRZ

13.5 Resumindo

Neste capítulo, aprendemos:

- A quebrar funções em funções menores;
- A usar constantes para evitar números mágicos repetidos;
- Ifs ternários.

Capítulo 14

Exercícios

Agora é sua vez de praticar!

14.1 Jogo de adivinhação

1) Mude o loop principal do jogo para um `do-while`, igual fizemos no jogo de forca.

14.2 Jogo de forca

1) Não permitir que o usuário digite nada além de letras maiúsculas. *Dica: Imprima a letra 'A' e a letra 'Z' como inteiros (usando* `%d`*) e descubra seus números. Em seguida, faça um* `if` *proibindo o caractere digitado de ser fora desse intervalo.*

2) Não permitir que o usuário insira uma palavra que já exista no arquivo de palavras.

3) Só possibilitar a inserção de uma nova palavra caso o jogador ganhe o jogo.

4) Pergunte ao usuário o nível de dificuldade que ele quer jogar. De acordo com o nível, você tem mais ou menos chutes.

5) A função `enforcou()` tem código repetido. Você pode fazer uso da função `jachutou()` e diminuir a duplicação de código.

6) Você deve ter reparado que, ao adicionarmos a décima palavra, teremos um problema. Isso porque sempre escrevemos em cima do que já está no arquivo. E como "10" é maior do que "9", temos um byte a mais, e o arquivo fica desposicionado (o *enter* some). Para resolver esse problema, combine que o número será sempre escrito com 4 algarismos (0001, 0009, 0021, e assim por diante). Assim, garantimos que nosso programa funcionará bem para até 9999 palavras. Para que isso funcione no `printf`, passamos `"%04d"` como máscara. Faça essa alteração no jogo.

7) Ao final do jogo, pergunte o nome do usuário e salve-o em um arquivo `ranking.txt`, junto com a sua pontuação.

14.3 Outros desafios

1) Escreva uma função que receba dois inteiros, `a` e `b`, e calcule a potência `a^b` ou seja, `a` elevado a `b`.

2) Escreva um programa que peça um número ao usuário e, com esse número, faça o programa escrever um arquivo `tabuada.txt` com a tabuada de 1 até 20 daquele número. Supondo o número 2 de entrada, a saída deve ser no formato `2 x 1 = 2`, e assim por diante.

3) Dados *n* e uma sequência de *n* números inteiros, determinar a soma dos números pares.

4) Escreva uma função que nos diz se um número é primo ou não.

5) Escreva um programa que recebe 2 números e calcula a soma dos números que são primos no intervalo.

6) Escreva uma função que receba um array de inteiros e some todos os elementos dentro desse array.

Capítulo 15

Jogo Foge-foge

Nosso próximo jogo será batizado de **Foge-foge**. Ele é muito similar ao famoso Pac-Man, que provavelmente você já jogou. Nosso herói passeia pelo

mapa, mas precisa ter cuidado com os fantasmas. Afinal, se algum encostar nele, você perde o jogo. E, cuidado, os fantasmas andam de maneira aleatória!

No entanto, o nosso herói também tem seus truques. Se ele tiver a pílula mágica, ele poderá explodir muitos fantasmas de uma só vez.

Fazer um jogo desses não é fácil. Aqui aprendemos muita coisa nova, como:

- Criar, varrer a manipular matrizes (ou arrays bidimensionais);
- Alocar memória dinamicamente;
- Criar estruturas (tipos) próprias;
- Fazer funções recursivas;
- Usar funções mais avançadas de manipulação de strings e de memória;
- Usar ponteiros de ponteiros;
- Usar outras diretivas de compilação.

15.1 Como nosso jogo vai ficar?

O desenho a seguir mostra como será a saída final do nosso jogo. O Foge-foge anda pelo mapa, usando as teclas `A, S, D, W`. Ele precisa desviar das paredes e tentar destruir os fantasmas. Se você já jogou Counter Strike, vai se sentir em casa!

Sempre que o herói andar, teremos um novo mapa impresso.

```
Pílula: NÃO
............................................................
............................................................
............................................................
............................................................
...... .-.   .-.   .-.  ......                        ......
......| OO| | OO| | OO| ......                        ......
......|   | |   | |   | ......                        ......
......'^^^' '^^^' '^^^' ......                        ......
```

```
......                                                ......
......                                                ......
......                                                ......
......                                                ......
...... .-.   .-.                     .--. ......      ......
......| OO| | OO|                   / _.-'......      ......
......|   | |   |                   \  '-.......      ......
......'~~~' '~~~'                    '__' ......      ......
......            .-.                     ......      ......
......           | OO|                    ......      ......
......           |   |                    ......      ......
......           '~~~'                    ......      ......
............................................................
............................................................
............................................................
............................................................
```

15.2 É hora de começar!

Está pronto para desenvolver esse jogo? **Mãos à obra!**

Capítulo 16
Matrizes

O primeiro passo para a implementação do Foge-foge é pensar no mapa em que nosso herói andará. Nele, precisamos representar tudo o que pode existir por lá, como: paredes, monstros, o próprio herói, e lugares pelos quais ele pode passar.

Podemos representar um mapa por meio de várias linhas e colunas. Em cada posição, teríamos um caractere que representaria o que tem ali. Veja o exemplo do mapa a seguir, no qual o ponto (`.`) significa "caminho livre", o arroba (`@`) representa nosso herói e os caracteres `|` e `-` seriam as paredes:

```
0123456789
0 |--------|
1 |...|..-.|
2 |..-|.@..|
3 |......-.|
4 |--------|
```

Nosso herói que está na linha 2 da coluna 6 pode andar para esquerda, direita, cima ou baixo. Mas não pode andar duas vezes seguidas para a esquerda, por exemplo, pois há uma parede ali.

Podemos representar cada linha desse mapa por meio de um array de chars. Como cada linha tem 10 posições, então um array de 10 posições resolveria. E como temos 5 linhas, precisamos de 5 desses arrays para representar todo o mapa:

```
char linha1[10];
char linha2[10];
char linha3[10];
char linha4[10];
char linha5[10];
```

Claramente essa não é uma boa solução. Imagine se tivéssemos um mapa com 50 linhas? Precisaríamos ter 50 variáveis dessas. Manipulá-las não seria fácil. A solução para isso é deixar de usar vetores e passar a usar **matrizes**. Matrizes, assim como na matemática, é uma estrutura retangular de dados, ou seja, nosso "linhas e colunas".

Fig. 16.1: Matrizes

Se quisermos uma matriz de chars de 5x10, declaramos uma variável para tal. Para declararmos matrizes em C, basta repetirmos duas vezes o conjunto de `[]`; um para linhas e outro para colunas, respectivamente:

```
char mapa[5][10];
```

Acessar e alterar valores das posições é análogo ao uso de vetores, com a diferença que agora passamos ambas as posições:

```
// armazenando na posição 0x0
mapa[0][0] = '|';

// imprimindo a posição 2x3
printf("%c", mapa[2][3]);
```

Para que nosso jogo fique interessante, vamos lê-lo de um arquivo. Nosso programa deve ler linha a linha e salvá-lo na matriz `mapa`. Vamos começar abrindo o arquivo, do jeito que já conhecemos:

```
FILE* f;

f = fopen("mapa.txt", "r");
if(f == 0) {
    printf("Erro na leitura do mapa");
    exit(1);
}
```

Agora precisamos mudar a declaração do nosso mapa. Apesar dele ter 10 caracteres na linha, precisamos lembrar que, quando usarmos o `fscanf()`, passando uma string como máscara, ele colocará um `\n` ao final. Podemos também usar as funções de leitura que vimos no final do capítulo 12, como a `fread()`, e ler byte a byte, mas isso nos daria um trabalho desnecessário. Vamos declarar o array com uma posição a mais em cada linha, apenas para guardar o *enter*:

```
char mapa[5][10+1];
```

Agora, basta lermos linha a linha. Como sabemos que temos 5 linhas exatas, basta fazermos um `for`. No capítulo 9, discutimos que um array/vetor é simplesmente um ponteiro que aponta para uma posição da memória que tem espaços vazios para guardar as *n* posições declaradas.

Uma matriz é uma abstração ainda maior. Ela é um ponteiro que aponta para uma lista de arrays. Como arrays são ponteiros, logo, uma matriz é um ponteiro que aponta para uma lista de ponteiros.

Ou seja, `mapa` é um ponteiro que aponta para outros ponteiros. Isso significa que `mapa[0]` é um ponteiro que aponta para um array, `mapa[1]` é um ponteiro que aponta para outro array, e assim por diante. Veja na imagem a seguir o funcionamento de uma matriz:

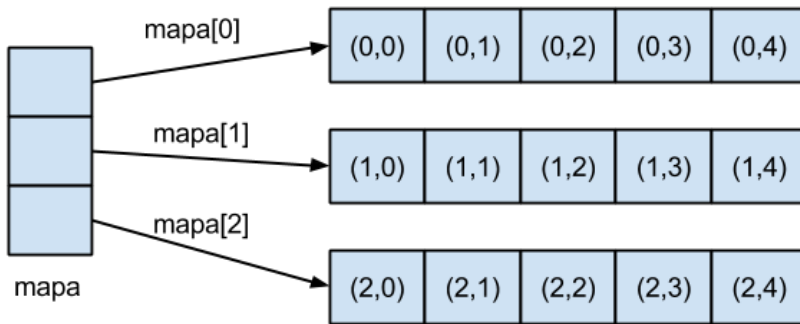



Fig. 16.2: Matrizes e ponteiros

16.1 Ponteiros de ponteiros

Em código, nada nos impede de declararmos uma matriz e depois declarar ponteiros que apontam para cada um dos vetores dentro dessa matriz. Veja o código a seguir. `linha0[0]` é igual a `numeros[0][0]`, afinal tanto `linha0` quanto `numeros[0]` apontam para o mesmo endereço de memória.

```
int numeros[5][10];

int* linha0 = numeros[0];
int* linha1 = numeros[1];
int* linha2 = numeros[2];
int* linha3 = numeros[3];
int* linha4 = numeros[4];

// os números serão iguais, afinal
// ambos ponteiros apontam para o mesmo
// endereço de memória
printf("%d %d", linha0, numeros[0]);
```

Ou seja, nesse loop que faremos, guardaremos a primeira linha no array `mapa[0]`, a segunda linha no array `mapa[1]`, e assim por diante. Veja que nesse código, declaramos `int*` para representar cada array que está dentro

da matriz. Mas e se quiséssemos um novo ponteiro para apontar para a matriz como um todo? Como ela é um ponteiro que aponta para outros ponteiros de inteiro, a declaração é `int**`, ou seja, duas estrelas para representar "ponteiro de ponteiro":

```
int numeros[5][10];

// o ponteiro copia é idêntico ao
// ponteiro numeros... ambos apontam
// para uma lista de ponteiros de inteiros
int** copia = numeros;

// as duas operações abaixo são idênticas
numeros[0][0] = 10;
copia[0][0] = 10;
```

Ou seja, enquanto um array é um ponteiro para uma lista de posições de memória que estão livres para guardar o tipo escolhido, uma matriz é um ponteiro que aponta para uma lista de ponteiros, que apontam para posições de memória livres para guardar algo.

Com isso em mãos, podemos agora fazer o loop para capturar as linhas do arquivo. Apontamos `mapa[i]` como o lugar onde os caracteres devem ser colocados:

```
for(int i = 0; i < 5; i++) {
    fscanf(f, "%s", mapa[i]);
}
```

Podemos imprimir o mapa de maneira simples, por enquanto:

```
for(int i = 0; i < 5; i++) {
    printf("%s\n", mapa[i]);
}
```

Por fim, precisamos fechar o arquivo:

```
fclose(f);
```

Pronto. Já estamos lendo a matriz de mapa de um arquivo. Mas podemos melhorar ainda mais.

Está perdido?

O código até o momento está em http://pastebin.com/QxtGZR4B

Ponteiros de função

Acredite ou não, podemos ter ponteiros que apontam para funções também. Entretanto, esse é um assunto que não é abordado neste livro, devido à sua complexidade.

16.2 Alocação dinâmica de memória

Mas e se quisermos mapas com tamanhos diferentes, para, por exemplo, diferenciar o nível do jogo? Quanto mais difícil, maior o mapa. O grande problema é: como variar o tamanho da matriz?

Uma possível estratégia seria ter uma matriz gigante que suportasse o maior mapa possível. Por exemplo, uma matriz `mapa[100][100]`. Nosso programa olharia apenas para a quantidade de linhas e colunas daquele mapa em particular, que será sempre menor que 100x100.

Aqui, tomaremos outra decisão: a de gerar um array (ou matrizes) de tamanhos aleatórios. Leremos o artigo, descobriremos o tamanho do mapa e, então, declararemos uma matriz com o tamanho ideal.

Para fazer isso, precisamos mudar a maneira de declararmos nosso array, alocando a memória necessária em tempo de execução. Para isso, usaremos a função `malloc()`. Ela recebe como parâmetro a quantidade de bytes que precisa ser alocado, e nos devolve um ponteiro para o primeiro byte alocado.

O código a seguir, por exemplo, aloca um único byte. Como ele nos devolve um ponteiro, o seu retorno pode ser armazenado em um ponteiro de `char`:

```
char* letra = malloc(1);
*letra = 'M';
```

Se quiséssemos guardar um `int`, precisaríamos de mais de 1 byte; afinal, inteiros são representados por 4 bytes. O código seria algo como:

```
int* numero = malloc(4);
*numero = 20;
printf("%d", *numero);
```

Mas como a quantidade de bytes que um inteiro ocupa pode variar de plataforma para plataforma, é melhor dizermos ao compilador para usar o tamanho do inteiro daquela plataforma. Para isso, usamos a instrução `sizeof(int)`, que nos retorna o tamanho correto:

```
int* numero = malloc(sizeof(int));
*numero = 20;
printf("%d", *numero);
```

Como o `malloc()` simplesmente aloca a quantidade de bytes desejada e nos devolve um ponteiro, para declararmos um array de maneira dinâmica, basta apenas passarmos a quantidade de bytes corretos. Por exemplo, se quisermos um array de inteiros de 10 posições, precisamos passar 40 para ele (4 bytes por inteiro vezes 10 inteiros). Com esse número, sabemos que podemos manipular um ponteiro como se fosse um array:

```
// calcula o tamanho de bytes
int colunas = 10;
int memoria = sizeof(int) * colunas;

// aloca memória suficiente para o array
int* numeros = malloc(memoria);

// usa o ponteiro como se fosse um array
numeros[0] = 10;
printf("%d", numeros[2]);
```

Já para uma matriz, temos mais de uma abordagem. Imagine uma matriz de 5 linhas por 10 colunas. Precisamos primeiro alocar um array para guardar 5 ponteiros de inteiros. Depois, alocar dez arrays de 5 arrays de 10 posições, e guardar cada um desses arrays nos ponteiros de inteiros declarados anteriormente.

Veja em código:

```
int** matriz;

int linhas = 5;
int colunas = 10;

// alocando espaço para as linhas,
// que guardam ponteiro de inteiro.
matriz = malloc(sizeof(int*) * linhas);

// agora, para cada linha, alocamos
// espaço para um array com 10 ("colunas") posições.
for(int i = 0; i < linhas; i++) {
    matriz[i] = malloc(sizeof(int) * colunas);
}

// agora podemos usar 'matriz' como uma matriz
matriz[2][3] = 10;
```

É isso o que faremos. Vamos recomeçar nosso código já o separando em funções (código organizado, sempre). Vamos declarar algumas variáveis globais que serão importantes ao longo do jogo: o tamanho do mapa e o mapa em si.

```
#include <stdio.h>
#include <stdlib.h>

char** mapa;
int linhas;
int colunas;
```

Nosso `mapa.txt` também terá um formato diferente. A primeira linha terá dois inteiros, indicando o tamanho do mapa que virá a seguir. No mapa atual, a primeira linha seria `5 10`:

```
5 10
|--------|
|...|..-.|
```

```
|..-|.@..|
|......-.|
|--------|
```

Ler isso é fácil, dado que sabemos usar bem a função `fscanf()`. Usaremos esses números para alocar o tamanho do mapa de maneira dinâmica, que será feito na função `alocamapa()`:

```
void lemapa() {
    FILE* f;
    f = fopen("mapa.txt", "r");
    if(f == 0) {
        printf("Erro na leitura do mapa");
        exit(1);
    }

    fscanf(f, "%d %d", &linhas, &colunas);
    alocamapa();

    for(int i = 0; i < linhas; i++) {
        fscanf(f, "%s", mapa[i]);
    }

    fclose(f);
}
```

A função `alocamapa()` contém um código bastante similar ao que usamos para entender alocação dinâmica. A diferença é que agora temos ponteiros para `char`. Apesar de sabermos que `char` é armazenado sempre em 1 bytes, vamos usar `sizeof()` para nunca termos problema. Lembre-se de que precisamos adicionar mais 1 byte, para guardar o *enter* que é salvo pelo `fscanf()`:

```
void alocamapa() {
    mapa = malloc(sizeof(char*) * linhas);

    for(int i = 0; i < linhas; i++) {
        mapa[i] = malloc(sizeof(char) * colunas + 1);
    }
}
```

Nossa `main` por enquanto apenas lerá o mapa e o imprimirá, para termos a certeza de que tudo está funcionando como gostaríamos:

```
int main() {

    lemapa();

    for(int i = 0; i < linhas; i++) {
        printf("%s\n", mapa[i]);
    }
}
```

Para também não termos problemas com ordem da declaração das funções, vamos desde já colocar todas as assinaturas dentro do `fogefoge.h`, que é importado no começo do nosso código-fonte:

```
// fogefoge.h
void alocamapa();
void lemapa();
```

Pronto. Nosso jogo agora é capaz de ler mapas de qualquer tamanho. Porém, ainda temos um problema. Você se lembra que, no capítulo 12, discutimos que todo recurso aberto deve ser fechado? Ou seja, se abrimos um arquivo, precisamos fechá-lo.

O mesmo acontece com memória que é alocada dinamicamente. Sempre que usamos `malloc()`, o sistema operacional nos reserva um pedaço da memória e não permite que nenhum outro programa encoste nele. No entanto, nosso programa é o responsável também por liberar essa memória de volta ao SO. Fazemos isso por meio da função `free()`. Ou seja, para cada `malloc()`, precisamos de um `free()` ao final.

Veja a função `liberamapa()`, que navega pela matriz, liberando cada um dos arrays alocados dinamicamente. Ao final, libera também a matriz como um todo:

```
void liberamapa() {
    for(int i = 0; i < linhas; i++) {
        free(mapa[i]);
    }
```

```
    free(mapa);
}
```

Agora, basta a invocarmos ao final do nosso programa, quando temos a certeza de que o mapa não será mais utilizado:

```
int main() {

    lemapa();

    for(int i = 0; i < linhas; i++) {
        printf("%s\n", mapa[i]);
    }

    liberamapa();
}
```

Neste momento, ao rodarmos o programa, temos o mapa como saída:

```
|--------|
|...|..-.|
|..-|.@..|
|......-.|
|--------|
```

> **Está perdido?**
>
> O código até o momento está em http://pastebin.com/HyxpNDPh

16.3 Resumindo

Neste capítulo, aprendemos:

- A declarar e manipular matrizes;
- O que são ponteiros de ponteiros;

- Que matrizes são, no fim, ponteiros de ponteiros;
- A declarar memória dinamicamente, usando `malloc()`;
- Que a instrução `sizeof()` nos devolve o tamanho em bytes de um tipo específico naquela plataforma;
- Que devemos liberar a memória alocada dinamicamente ao fim, com `free()`.

Capítulo 17

Structs

Nosso próximo passo é fazer o herói andar. Como todo jogo, usaremos a combinação de teclas `A, S, D, W`. A ideia será a seguinte: exibiremos o mapa, leremos uma entrada do teclado, faremos as atualizações necessárias no mapa e repetiremos tudo de novo, até que o jogador ganhe ou perca.

Vamos começar com nosso loop principal, que será parecido com o do nosso jogo de forca. Usaremos a função `scanf()`, do jeito com que você já está acostumado. Lembre-se do espaço em branco para que o *enter* seja ignorado:

```
int main() {

    lemapa();

    do {
```

```
        imprimemapa();

        char comando;
        scanf(" %c", &comando);
        move(comando);

    } while (!acabou());

    liberamapa();
}
```

A função `acabou()`, como o seu próprio nome diz, nos informará se o jogo acabou ou não. Por enquanto, vamos fazer uma função "falsa", que sempre nos retorna `0`. Ou seja, o jogo nunca acaba:

```
int acabou() {
    return 0;
}
```

A função `imprimemapa()` é também bastante simples. Ela apenas imprime o mapa, do jeito que ele está na matriz:

```
void imprimemapa() {
    for(int i = 0; i < linhas; i++) {
        printf("%s\n", mapa[i]);
    }
}
```

Com o comando em mãos, podemos mover o herói para a direção que o jogador pediu. O primeiro passo do algoritmo deve ser localizar o herói. Para isso, não temos como fugir de varrer toda a matriz, usando dois loops encadeados:

```
void move(char direcao) {
    int x;
    int y;

    for(int i = 0; i < linhas; i++) {
        for(int j = 0; j < colunas; j++) {
```

```
            if(mapa[i][j] == '@') {
                x = i;
                y = j;
                break;
            }
        }
    }

}
```

Em seguida, modificaremos o mapa de acordo com a direção escolhida: `A` vai para a esquerda, `D` para a direita, `S` para baixo e `W` para cima. Se o jogador andou para esquerda, devemos deslocar o herói para lá, ou seja, para a `linha, coluna-1`. Se ele andou para direita, o herói vai para `linha, coluna+1`; se para cima, vai para `linha-1, coluna`; e se para baixo, para `linha+1, coluna`. Ao final, devemos colocar como vazio a posição que o herói estava antes. Resolvemos isso com um `switch`:

```
void move(char direcao) {
    // os fors aqui...

    switch(direcao) {
        case 'a':
            mapa[x][y-1] = '@';
            break;
        case 'w':
            mapa[x-1][y] = '@';
            break;
        case 's':
            mapa[x+1][y] = '@';
            break;
        case 'd':
            mapa[x][y+1] = '@';
            break;
    }

    mapa[x][y] = '.';
}
```

Se você rodar o jogo, verá que ele já começa a se parecer com o que queremos. Conseguimos movimentar nosso personagem para todas as direções. Mas ainda estamos longe de terminá-lo: se digitarmos uma letra inválida, o herói desaparece. Ele também atravessa paredes. Temos muito a corrigir.

Está perdido?

O código até o momento está em http://pastebin.com/LSAFA6BV

17.1 Definindo uma struct

Repare que, sempre que usamos o `mapa`, utilizamos também as variáveis `linhas` e `colunas`. Afinal, sempre precisamos delas para navegar na matriz. Como essas variáveis são globais, não temos problemas para usá-las. Mas e se precisássemos passá-las como parâmetros para alguma função? Seria trabalhoso; teríamos muitos parâmetros.

Pela primeira vez temos um conjunto de variáveis que *não fazem sentido separadas*. A matriz só faz sentido se tiver uma linha e coluna junto. Precisamos agrupá-las, e garantir que elas sempre serão "transportadas" juntas. Para tal, usaremos uma **struct**.

Estruturas (ou, em C, `struct`) são uma maneira que temos para agrupar variáveis. Ou seja, definiremos que a estrutura `mapa` contém uma matriz (o ponteiro de ponteiro de `char`), uma variável para guardar a quantidade de linhas (`int`) e outra para guardar a quantidade de colunas (também `int`). Com essa estrutura definida, podemos declarar variáveis que a seguem. Dessa forma, garantiremos sempre que mapas têm uma matriz, e a quantidade de linhas e colunas.

Veja a declaração da `struct` a seguir, que colocamos dentro do nosso `fogefoge.h`. Dentro dele, declaramos a matriz e os dois inteiros para guardar as suas dimensões:

```
struct mapa {
    char** matriz;
```

```
    int linhas;
    int colunas;
};
```

Como dissemos anteriormente, com essa struct em mãos, podemos fazer uso dela e declarar variáveis desse tipo. Mas, diferente de um inteiro ou um char que utilizamos até então, ela tem variáveis dentro de si. Para acessá-las, usamos ponto (`.`). Veja o trecho de código a seguir, por exemplo, no qual definimos um mapa e setamos os valores para `linhas` e `colunas`. Repare na palavra `struct` no começo da declaração:

```
struct mapa facil;

char tabuleiro[10][20];
facil.matriz = &tabuleiro;
facil.linhas = 10;
facil.colunas = 20;

printf("O mapa tem tamanho %d x %d", facil.linhas,
        facil.colunas);
```

Vamos fazer as devidas alterações no código, usando agora a `struct` que definimos. A começar pelas variáveis globais, que são substituídas:

```
struct mapa m;
```

Em todo lugar que fazia uso de `linhas` ou `colunas`, precisamos trocar para `m.linhas` e `m.colunas`. Em todo lugar que tínhamos a matriz `mapa`, agora torna-se necessário usar `m.matriz`.

Um detalhe importante é perceber precedência de operadores. Por exemplo, dentro da função `lemapa()`, temos o `fscanf()`, que agora ficou parecido com o código a seguir:

```
fscanf(f, "%d %d", &m.linhas, &m.colunas);
```

A instrução `&m.linhas` é processada corretamente. Passamos o endereço da variável `linhas` que está dentro da `struct m`. Uma outra interpretação dessa instrução seria que o `&` é para a variável `m`, ou seja, gostaríamos

do endereço de memória da estrutura `m` como um todo. Para evitar essa possível ambiguidade, é comum fazermos uso de parênteses para deixar clara a ordem de precedência:

```
fscanf(f, "%d %d", &(m.linhas), &(m.colunas));
```

A função `move()`, com todas as alterações, fica como a que segue:

```
void move(char direcao) {
    int x;
    int y;

    for(int i = 0; i < m.linhas; i++) {
        for(int j = 0; j < m.colunas; j++) {
            if(m.matriz[i][j] == '@') {
                x = i;
                y = j;
                break;
            }
        }
    }

    switch(direcao) {
        case 'a':
            m.matriz[x][y-1] = '@';
            break;
        case 'w':
            m.matriz[x-1][y] = '@';
            break;
        case 's':
            m.matriz[x+1][y] = '@';
            break;
        case 'd':
            m.matriz[x][y+1] = '@';
            break;
    }

    m.matriz[x][y] = '.';
}
```

Apesar da sensação de que agora temos muito mais código escrito, usar `structs` para agrupar dados que devem ficar perto é uma boa prática. Você verá que, em breve, deixaremos de usar variáveis globais e começaremos a passar essas `structs` para lá e para cá. Como elas agrupam tudo o que precisamos, fica fácil: a função receberá apenas um ponteiro para a estrutura.

Repare que, sempre que precisamos declarar uma variável do tipo da estrutura, é necessário usar a palavra `struct` antes como `struct mapa m`. Podemos melhorar isso, fazendo uso da instrução `typedef`. Ela nos possibilita dar um outro nome a algum tipo já existente.

Por exemplo, vamos apelidar `struct mapa` de `MAPA`, dentro do nosso header file:

```
typedef struct mapa MAPA;
```

Graças ao `typedef`, agora podemos declarar variáveis do tipo `MAPA` sem a palavra `struct`. Veja:

```
MAPA m;
```

Está perdido?

O código feito até o momento está em http://pastebin.com/YqyZdN6B

17.2 Ponteiros para structs

Você reparou também que a maioria das funções que escrevemos até então tem a palavra "mapa" em alguma parte do nome? A estrutura `mapa`, bem como as funções `alocamapa()`, `lemapa()`, `liberamapa()` e `imprimemapa()` fazem algo relacionado ao mapa. As funções restantes `acabou()`, `move()` e `main()` apenas fazem uso da estrutura e/ou funções do mapa.

Podemos pensar em isolar todas as funções em um arquivo específico, por exemplo, `mapa.c`. Mas o que ganhamos com isso? Reúso e manutenibilidade. Ter um arquivo dedicado somente a lidar com mapas nos possibilita

utilizá-lo em outras aplicações (mapas podem ser úteis em muitos outros jogos). Além disso, sabemos que, se algo der errado com mapas, muito provavelmente esse código está dentro do arquivo específico.

A discussão aqui é similar à que fizemos quando chegamos à conclusão de que funções eram úteis para isolar e reutilizar código. Aqui, a mesma coisa, mas em um nível maior. Teremos um arquivo que agrupará um conjunto de funções que têm uma relação entre si.

Vamos criar o arquivo `mapa.h`, com a definição da `struct` e das assinaturas das funções que lidam com mapas:

```
struct mapa {
    char** matriz;
    int linhas;
    int colunas;
};

typedef struct mapa MAPA;

void alocamapa();
void lemapa();
void liberamapa();
void imprimemapa();
```

Já o arquivo `mapa.c` contém a implementação dessas funções:

```
// mapa.c
#include <stdio.h>
#include <stdlib.h>
#include "mapa.h"

void lemapa() {
    // código aqui
}

void alocamapa() {
    // código aqui
}
```

```
void liberamapa() {
    // código aqui
}

void imprimemapa() {
    // código aqui
}
```

Já no `fogefoge.c`, incluímos o header dos mapas. Afinal, precisamos dessas funções disponíveis também nesse arquivo. E, obviamente, retiramos as funções de mapa, que agora estão em seu arquivo particular:

```
#include <stdio.h>
#include <stdlib.h>
#include "fogefoge.h"
#include "mapa.h"
```

Vamos compilar nosso programa. A linha de compilação agora precisa contemplar ambos os arquivos. Para isso, basta passar a lista deles para o `gcc`:

```
gcc fogefoge.c mapa.c -o fogefoge.out
```

Entretanto, o compilador nesse momento nos devolve muitos erros. A maioria deles reclamando sobre a não existência da variável `m` no arquivo `mapa.c`. E faz sentido: a variável `m` está declarada em outro arquivo.

Variáveis, mesmo que globais, são visíveis apenas no arquivo em que foram declaradas. Ou seja, a variável `m` é visível apenas dentro do `fogefoge.c`. E faz sentido ela ser declarada lá, uma vez que ela representa um mapa do nosso jogo. Não é algo genérico para colocarmos dentro do `mapa.c`.

Ainda assim, queremos separar as funções de mapa do resto. Portanto, não teremos outra solução a não ser fazer as funções que lidam com mapa e receber um `mapa` como parâmetro. Vamos pegar, por exemplo, a função `alocamapa()` e fazer isso:

```
void alocamapa(mapa m) {
    m.matriz = malloc(sizeof(char*) * m.linhas);
```

```
    for(int i = 0; i < m.linhas; i++) {
        m.matriz[i] = malloc(sizeof(char) * m.colunas + 1);
    }
}
```

O código agora compilará. Porém, da forma como ele está escrito, teremos um problema. Para entendê-lo, criaremos um outro programa bastante simples, que ilustrará o que precisamos.

O código a seguir declara uma simples estrutura `quadrado` com dois inteiros. A função `dobra()` dobra os valores de `x` e `y`. A função `main()` imprime os valores dos quadrados antes e depois de eles serem dobrados.

```
#include <stdio.h>

struct quadrado {
    int x;
    int y;
};

typedef struct quadrado QUADRADO;

void dobra(QUADRADO q) {
    q.x = q.x * 2;
    q.y = q.y * 2;
}

int main() {
    QUADRADO q1;

    q1.x = 20;
    q1.y = 35;

    printf("%d %d\n", q1.x, q1.y);
    dobra(q1);
    printf("%d %d\n", q1.x, q1.y);
}
```

Mas, ao rodarmos o programa, temos uma saída que não esperamos: os

resultados são iguais, antes e depois da função de dobra:

```
20 35
20 35
```

A razão do resultado é fácil de ser entendida. E você, na verdade, já sabe, mas pode ter se confundido por causa da `struct`. Veja o código a seguir, mais simples ainda, que, em vez de dobrar o valor dentro de uma `struct`, dobra o valor de uma variável `n`:

```
void dobra(int n) {
    n = n * 2;
}

int main() {
    int n = 10;
    printf("%d", n);
    dobra(n);
    printf("%d", n);
}
```

A saída do programa é `10 10`. Veja que a variável `n` do método `main` é diferente da variável `n` do método `dobra`. A máquina, no momento em que invocamos o método `dobra()`, gera uma cópia da variável `n` que está em `main` e a passa para a variável `n` da função `dobra()`. Ou seja, passagem por cópia. O conteúdo é **copiado** para o parâmetro. Já vimos isso durante o capítulo 11. Foi exatamente por isso que optamos por usar ponteiros, aliás.

O mesmo aconteceu com a `struct`. Ao passarmos a variável `q1` declarada na `main` para a função `dobra`, a máquina passou apenas uma cópia da `struct` original. As mudanças, portanto, foram feitas na cópia. Se quisermos modificar a `struct` original, da mesma forma que se quisermos modificar qualquer outra variável original, precisamos lidar com ponteiros.

A função `alocamapa()` deve receber um ponteiro de mapa, para que suas alterações sejam feitas sempre na posição de memória da variável original. Ou seja, passagem por referência:

```
void alocamapa(mapa* m) {
    // ...
}
```

Agora precisamos mudar o corpo da função. Afinal, antes fazíamos `m.linhas` para acessar o conteúdo da variável `linhas`, dentro de `m`. Mas agora, `m` é um ponteiro para um mapa. Ou seja, precisamos primeiro ir para o conteúdo do ponteiro, para depois usar a variável `linhas`. Em código, `(*m).linhas`. Veja a primeira linha dessa função, modificada:

```
(*m).matriz = malloc(sizeof(char*) * (*m).linhas);
```

Dado que isso é uma operação bastante comum (acessar primeiro o conteúdo do ponteiro e depois uma variável qualquer dentro da `struct`), a linguagem C possui um açúcar sintático que nos ajuda a escrever a mesma coisa de maneira mais simples. Para isso, usamos uma seta (`->`). A seta faz a mesma coisa que o "estrela-ponto" que usamos no código anterior. Por exemplo, `m->linhas`. Veja a primeira linha:

```
m->matriz = malloc(sizeof(char*) * m->linhas);
```

Veja a função `alocamapa()` inteira:

```
void alocamapa(MAPA* m) {
    m->matriz = malloc(sizeof(char*) * m->linhas);

    for(int i = 0; i < m->linhas; i++) {
        m->matriz[i] = malloc(sizeof(char) * m->colunas + 1);
    }
}
```

Todas as outras funções do arquivo `mapa.c` devem seguir a mesma abordagem: receber um ponteiro de mapa como parâmetro e fazer uso da seta para manipular esse mapa. As funções agora têm essa assinatura:

```
struct mapa {
    char** matriz;
    int linhas;
    int colunas;
};

typedef struct mapa MAPA;
```

```
void alocamapa(MAPA* m);
void lemapa(MAPA* m);
void liberamapa(MAPA* m);
void imprimemapa(MAPA* m);
```

A ordem importa?

No header file, você precisa colocar o `struct mapa` antes das declarações das funções. Afinal, na assinatura da função, usamos o tipo `MAPA`, e ele precisa ser declarado antes. Veja o código anterior com atenção.

Não podemos esquecer também de passar o parâmetro para a função `alocamapa()`, invocada dentro da `lemapa()`:

```
void lemapa(MAPA* m) {
    // ...

    fscanf(f, "%d %d", &(m->linhas), &(m->colunas));
    alocamapa(m);

    // ...
}
```

De volta ao `fogefoge.c`, precisamos agora passar o endereço de memória da variável `m` para as funções (usando `&`). Nesse momento, as alterações concentram-se na função `main`:

```
int main() {

    lemapa(&m);

    do {
        imprimemapa(&m);

        char comando;
        scanf(" %c", &comando);
```

```
        move(comando);

    } while (!acabou());

    liberamapa(&m);
}
```

Nosso jogo agora voltou a compilar e a funcionar. Repare que o comportamento é o mesmo que já tínhamos antes de começar toda essa refatoração. A diferença é que agora temos nosso código ainda mais fácil de ser lido e mantido.

Variáveis globais não são globais?

Variáveis declaradas como globais são globais para todo o programa. Mas, para que um arquivo enxergue uma variável global definida em outro arquivo, precisamos fazer uso da palavra `extern`.

Imagine, por exemplo, uma variável `int n;` definida no arquivo `a.c`. Para a usarmos no arquivo `b.c`, precisaríamos redeclará-la com `extern int n;`. Agora sim, a variável `n`, global, é visível por ambos arquivos. Repare que só um arquivo a realmente declara, e os outros deixam claro que ela é "externa".

Está perdido?

O código até o momento está em http://pastebin.com/PAz81Cda

17.3 Introdução à análise de algoritmos

Analise agora a função `move()`. Analisar funções é algo comum em nosso dia a dia de trabalho, uma vez que, se uma delas não funcionar bem ou levar muito tempo para executar, podemos ter problemas em nosso software.

Quando analisamos uma função, geralmente vemos o trabalho que ela tem em relação aos dados de entrada que ela recebe. Nesse caso, consideramos o `mapa` como seu dado de entrada. Veja que, sempre que invocamos a função, os dois `for`s garantem que toda a matriz é percorrida. Se o herói estiver na última posição inferior à direita, o algoritmo passeará por **todas** as posições dela.

Ou seja, quanto maior o mapa, mais lerda essa função será. Se fôssemos definir isso em termos matemáticos, diríamos que essa função tem a complexidade de, no máximo, `linhas * colunas`. Aqui não entraremos em detalhes sobre **análise de algoritmos**, que é uma disciplina fundamental em computação. No entanto, perceba desde já que, se você tem algoritmos que passeiam por todo um array ou matriz, quanto maior esse array, mais tempo essa função levará. Será que conseguimos "economizar" no número de vezes que varremos a matriz?

Repare que esses loops servem apenas para encontrar a posição atual do herói no mapa. Uma solução para isso é termos duas variáveis globais, que guardariam o X e Y do herói. Dessa forma, não precisaríamos buscá-lo o tempo todo; já teríamos essa informação salva.

Vamos declarar duas variáveis globais para isso:

```
int x;
int y;
```

Com essas duas variáveis, podemos eliminar os `for`s encadeados, e a função `move()` fica ainda menor. Nesse momento, repare que, além de menor, ela também é mais rápida: agora ela não varre toda a matriz para fazer seu trabalho. Independente do tamanho da matriz, ela gastará sempre o mesmo tempo para executar. Dizemos que essa função leva tempo constante para executar:

```
void move(char direcao) {

    switch(direcao) {
        case 'a':
            m.matriz[x][y-1] = '@';
            break;
```

```
        case 'w':
            m.matriz[x-1][y] = '@';
            break;
        case 's':
            m.matriz[x+1][y] = '@';
            break;
        case 'd':
            m.matriz[x][y+1] = '@';
            break;
    }

    m.matriz[x][y] = '.';
}
```

Precisamos agora encontrar o lugar ideal para encontrar a posição do herói pela primeira (e única) vez. Podemos fazer isso logo após a invocação da função `lemapa()` no método `main()`. Mas antes, vamos declarar outra estrutura. Perceba que sempre que há um X, há também um Y. Por que não ter a estrutura `posicao`? Como uma posição é relacionada a um mapa, essa estrutura deve ser declarada no header file de mapa:

```
struct posicao {
    int x;
    int y;
};

typedef struct posicao POSICAO;
```

Vamos agora criar a função que localiza o herói e guarda sua posição em uma estrutura dessas. Repare que recebemos o ponteiro de ambas as estruturas. Note também nas setas, usadas para armazenar os valores dentro da posição de memória para a qual os ponteiros apontam:

```
void encontramapa(MAPA* m, POSICAO* p, char c) {

    for(int i = 0; i < m->linhas; i++) {
        for(int j = 0; j < m->colunas; j++) {
            if(m->matriz[i][j] == c) {
                p->x = i;
```

```
                p->y = j;
                return;
            }
        }
    }

}
```

Precisamos agora declarar uma `POSICAO` que guardará a posição do nosso herói, e invocar a função `encontramapa()` logo após lermos o mapa:

```
// variáveis globais
MAPA m;
POSICAO heroi;

// ...

int main() {

    lemapa(&m);
    encontramapa(&m, &heroi, '@');

    // ...

}
```

Por fim, devemos fazer com que a função `move()` faça uso agora dos `X` e `Y` que estão dentro do `heroi`. Além disso, é necessário também atualizar essa variável sempre que o herói andar. Vamos fazê-lo dentro de cada opção do `switch`. Repare que fazemos a substituição da posição atual do herói por vazio no começo da função; afinal, já que alteramos os valores da variável `heroi`, após o `switch` a variável já aponta para a nova posição do herói.

```
void move(char direcao) {

    m.matriz[heroi.x][heroi.y] = '.';

    switch(direcao) {
        case 'a':
```

```
            m.matriz[heroi.x][heroi.y-1] = '@';
            heroi.y--;
            break;
        case 'w':
            m.matriz[heroi.x-1][heroi.y] = '@';
            heroi.x--;
            break;
        case 's':
            m.matriz[heroi.x+1][heroi.y] = '@';
            heroi.x++;
            break;
        case 'd':
            m.matriz[heroi.x][heroi.y+1] = '@';
            heroi.y++;
            break;
    }

}
```

Nosso jogo continua funcionando. Agora com um algoritmo ainda mais performático. Isso é, em alto nível, o que chamamos de *análise de algoritmo*. É olhar para ele e avaliar seu desempenho. Esse desempenho pode ser avaliado de diferentes maneiras: consumo de memória, tempo de processamento, espaço em disco etc.

Geralmente, estamos preocupados com o tempo de processamento. Antes, nossa função `move()` levava um tempo proporcional ao tamanho da matriz. Ou seja, quanto maior a matriz, mais tempo ele levava. Vimos ainda que, no pior caso, o tempo do algoritmo poderia ser proporcional ao número de linhas vezes o número de colunas (**linhas x colunas**), justamente porque ele era obrigado a passear por toda a matriz. Já na nova versão, o tempo de execução da função não varia de acordo com o tamanho da matriz. Ou seja, o tempo dela é sempre o mesmo: é constante.

É isso que você aprenderá na disciplina de análise de algoritmos. Obviamente, por lá você verá todo o ferramental matemático para avaliar e expressar o tempo de execução desses algoritmos. Não é o foco deste livro, claramente, mas perceba desde já a importância do tema.

Está perdido?

O código até o momento está em http://pastebin.com/WDnyN8kV

17.4 Resumindo

Neste capítulo, aprendemos:

- Por que é importante deixar perto dados que se relacionam;
- A definir structs;
- A declarar variáveis do tipo da struct;
- A usar o `typedef` para definir um nome de tipo e vinculá-lo a uma struct;
- A separar funções em diferentes arquivos;
- A compilar mais de um arquivo `.c` junto, para formar um só programa;
- A passar structs como referência;
- A usar o operador `->` para manipular ponteiros de structs;
- Que análise de algoritmos é uma importante área da computação.

Capítulo 18

Programando como um profissional

Está na hora de continuarmos nosso jogo. Afinal, ele é, de longe, o mais complicado dos três. A primeira coisa a fazer é acertar o movimento do nosso herói. Nesse momento, se apertarmos alguma tecla que não seja de movimentação, o herói some do mapa. Mais ainda, ele não pode andar por onde quiser: ele deve parar nas paredes e extremos do mapa.

Vamos começar ignorando qualquer comando que não seja de andar. Nesse momento do código, podemos fazer isso dentro da própria função `move()`. Um simples `if` resolve: se a letra não for `a`, `s`, `d`, `w`, então sai da função sem fazer nada.

Apesar de até então termos usado a palavra `return` para devolver valores em uma função, podemos usá-la para terminar a execução de um método

`void`. Nesses casos, a palavra `return` não vem acompanhada de nada. Repare que esse `if` é comprido e invertido. Ou seja, se a letra não for nem `a`, nem `w`, nem `s`, nem `d`, então sai da função:

```
void move(char direcao) {

    if(
        direcao != 'a' &&
        direcao != 'w' &&
        direcao != 's' &&
        direcao != 'd'
    ) return;

    // ...
}
```

A próxima validação é tratar o caso da parede e dos extremos. Nosso herói só pode andar se a casa na posição escolhida é um `.` (ponto) ou seja, vazio. Claro, se ele já estiver na última casa à direita, ele não pode ir mais para a direita; o mesmo para esquerda, cima e baixo. +

Para fazer esse `if`, precisaremos mudar um pouco nossa implementação. O cálculo da próxima posição é feita dentro do `switch`, onde calculamos a posição e já modificamos a matriz, colocando o herói. Agora precisamos validar antes de colocá-lo lá. Para isso, criaremos duas variáveis auxiliares que armazenarão a posição em que o herói deve ser colocado, sendo nelas que faremos as validações.

```
void move(char direcao) {

    if(
        direcao != 'a' &&
        direcao != 'w' &&
        direcao != 's' &&
        direcao != 'd'
    ) return;

    int proximox = heroi.x;
    int proximoy = heroi.y;
```

```
    switch(direcao) {
        case 'a':
            proximoy--;
            break;
        case 'w':
            proximox--;
            break;
        case 's':
            proximox++;
            break;
        case 'd':
            proximoy++;
            break;
    }

}
```

A validação é fácil. Sabemos que `proximox` deve ser menor do que a quantidade de linhas da matriz, e que `proximoy` deve ser menor do que a quantidade de colunas. Vamos escrever aqui o `if` ao contrário, pois usaremos a mesma estratégia com o caractere da direção: se `proximox` for maior ou igual à quantidade de linhas, então saímos da função. A mesma coisa para o `proximoy`:

```
if(proximox >= m.linhas)
    return;
if(proximoy >= m.colunas)
    return;
```

Repare que não abrimos chaves no `if`. Podemos fazer isso. Nesses casos, *a única coisa dentro do* `if` é a próxima instrução. Ou seja, o que fizemos anteriormente é o mesmo que fazeremos no código a seguir:

```
if(proximox >= m.linhas) {
    return;
}
if(proximoy >= m.colunas) {
    return;
}
```

A diferença é que essa ocupa mais linhas. Isso é um pouco de gosto de cada desenvolvedor. Na maioria das vezes, optamos por abrir chaves, uma vez que isso facilita a legibilidade. Mas nesse caso vamos deixar de usá-las; afinal, os `ifs` são muito parecidos e sequenciais. A falta das chaves não atrapalha a legibilidade.

A próxima etapa é garantir que a nova posição é vazia. Só assim ele poderá ir para lá. Basta adicionar mais um `if`, também invertido, que verifica se a posição contém um `.` (ponto):

```
if(m.matriz[proximox][proximoy] != '.')
    return;
```

Isso quer dizer que, se nosso código sobreviveu por todos esses `ifs`, então o herói está pronto para ir para essa posição. Precisamos agora fazer a troca: colocar o herói na nova posição, marcar a sua antiga posição como vazia, e acertar nossa variável `heroi` com as novas coordenadas:

```
m.matriz[proximox][proximoy] = '@';
m.matriz[heroi.x][heroi.y] = '.';
heroi.x = proximox;
heroi.y = proximoy;
```

E se fizéssemos sem os ifs invertidos?

Teríamos uma quantidade grande de `ifs` aninhados. Veja que a função só deve executar se a tecla digitada é uma direção, se `proximox` for menor que a quantidade de linhas, se `proximoy` for menor que a quantidade de colunas ou se a próxima posição é vazia.

Ou seja, teríamos muitos `ifs` aninhados, e o código ficaria mais ou menos como o mostrado na sequência. Veja que o código real que importa ficaria lá para a frente, com muitas indentações, o que dificultaria sua legibilidade. Por isso, é comum fazermos o contrário: se uma condição falha, saímos da função.

```
if(primeira regra está certa) {
    if(segunda regra está certa) {
        if(terceira regra está certa) {
            // ufa, agora sim vou fazer o que eu quero
            printf("Tudo é valido!");
        }
    }
}
```

Pronto. Nosso herói agora só anda por onde é realmente válido.

Está perdido?

O código até o momento está em http://pastebin.com/p8yTC1ca

18.1 Novamente, responsabilidades

Repare que, apesar de o andar do herói funcionar bem da forma que queríamos, nosso código não está perfeito. A função `move()` é muito grande e tem responsabilidades demais. Quando dizemos responsabilidades, estamos nos

referindo à quantidade de regras de negócio diferentes que está dentro dela. Veja que ela é responsável por saber se o caractere é uma direção válida, por calcular a próxima posição do herói, por saber se a nova posição está dentro do mapa e é válida, e por fazer as manipulações na matriz.

É simplesmente muita coisa para uma única função. Aliás, tem código aí que só tem relação com mapas, e já sabemos que tudo que diz respeito a mapas deveria estar no `mapa.c`. Esse tipo de situação é bastante comum no dia a dia. Afinal, estávamos muito preocupados em fazer a regra de negócio funcionar; acabamos não pensando na qualidade do código. É por isso que devemos sempre refletir sobre o estado atual do código, e melhorá-lo, se necessário.

Vamos começar a quebrar essa função. A primeira coisa que podemos fazer é levar esse `if` que verifica se a letra digitada é uma direção que é grande , para uma função própria. A função se chamará `ehdirecao(char)` e nos retornará um inteiro 0 ou 1. Note que precisamos inverter todo o `if`, pois agora a função nos retornará verdadeiro se o caractere é uma direção:

```
int ehdirecao(char direcao) {
    return
        direcao == 'a' ||
        direcao == 'w' ||
        direcao == 's' ||
        direcao == 'd';
}
```

A função `move()` já começa a ficar menor e mais fácil de ser entendida. Repare que, antes, o desenvolvedor precisava ler o `if` e entendê-lo. Agora não: o nome da função deixa bem claro o que ela faz. O desenvolvedor não precisa nem ler a implementação se não quiser:

```
void move(char direcao) {

    if(!ehdirecao(direcao))
        return;

    // ...
}
```

Podemos deixar o `switch` dentro da função. Afinal, calcular a posição é parte de "mover". Mas os dois próximos `ifs` podem ir para dentro das funções de mapa; eles nos dizem se uma determinada posição do mapa é válida ou não. Aqui, nossa estratégia será retornar 0, caso alguma das posições seja inválida; ou, no fim, retornar 1, dizendo que `x` e `y` estão dentro dos limites do mapa. Em código:

```
int ehvalida(MAPA* m, int x, int y) {
    if(x >= m->linhas)
        return 0;
    if(y >= m->colunas)
        return 0;

    return 1;
}
```

O próximo `if` nos diz se a próxima posição do mapa é vazia. Isso também pode ir para dentro do mapa. Uma simples função que nos devolve um booleano deixará nosso código muito mais claro. Como ela manipula mapa, logo, está dentro do `mapa.c`. Repare que invertemos a condição do `if` para que ela nos diga se aquela posição é ou não vazia:

```
int ehvazia(MAPA* m, int x, int y) {
    return m->matriz[x][y] == '.';
}
```

Os `ifs`, então, devem ser substituídos por invocações a essas funções. Veja que, apesar de termos diminuído pouco a quantidade de condicionais, esse código é muito mais fácil de ser entendido. Afinal, a palavra `ehvalida` deixa mais claro o que acontece do que aquele `if` grande. Precisamos ler menos, o que é bom:

```
if(!ehvalida(&m, proximox, proximoy))
    return;

if(!ehvazia(&m, proximox, proximoy))
    return;
```

Vamos agora colocar as próximas duas linhas, que são responsáveis por mover o que está em uma posição, para a nova posição. Essa também é uma função que deve estar no mapa e que provavelmente será utilizada mais à frente, pois teremos fantasmas no jogo, que também andarão para lá e para cá.

A função `andanomapa()` receberá 5 argumentos: um ponteiro de mapa, X e Y de origem, e X e Y de destino. Ela é bem simples. Primeiro, ela armazena o personagem (ou qualquer coisa) que estava na posição antiga, para em seguida, fazer a troca: coloca o personagem na posição de destino, e troca por vazio a posição antiga:

```
void andanomapa(MAPA* m, int xorigem, int yorigem,
    int xdestino, int ydestino) {

    char personagem = m->matriz[xorigem][yorigem];
    m->matriz[xdestino][ydestino] = personagem;
    m->matriz[xorigem][yorigem] = '.';

}
```

O código também fica mais simples no `move()`, já que, agora, basta invocar a nossa nova função que faz andar um personagem no mapa, e trocar os valores da posição do nosso `heroi`:

```
andanomapa(&m, heroi.x, heroi.y, proximox, proximoy);
heroi.x = proximox;
heroi.y = proximoy;
```

Pronto. A função `move()` agora é muito mais coesa. Ou seja, ela faz muito menos tarefas e delega para outras funções. Temos agora funções menores e mais fáceis de serem lidas, mantidas e reutilizadas. Isso é programar.

Está perdido?

O código até o momento está em http://pastebin.com/eCuugsgD

18.2 Novamente, constantes

Desde o primeiro jogo, discutimos os problemas de termos números mágicos espalhados pela aplicação. No momento, não temos números mágicos, mas sim "chars mágicos". Veja que temos em nosso código as letras `'a'`, `'d'`, `'s'`, `'w'`, que significam as possíveis direções, ou mesmo `'@'`, `'.'` e `'|'`, que indicam os possíveis personagens no mapa.

Vamos criar constantes para isso. Você já conhece as vantagens: maior legibilidade, já que uma constante tem um nome; e facilidade na manutenção, uma vez que basta alterar o valor da constante para a alteração propagar.

As constantes que dizem respeito aos mapas vão para o `mapa.h`:

```
#define HEROI '@'
#define VAZIO '.'
#define PAREDE_VERTICAL '|'
#define PAREDE_HORIZONTAL '-'
```

Já as constantes que dizem respeito ao jogo vão para o `fogefoge.h`:

```
#define CIMA     'w'
#define BAIXO    's'
#define DIREITA  'd'
#define ESQUERDA 'a'
```

Agora, basta fazermos uso delas em vez de termos esses caracteres fixos. A função `ehdirecao()`, por exemplo, fica ainda mais legível. Também usaremos essas constantes dentro do `switch`, na função `move()`. Repare que, se decidirmos mudar as teclas do jogo, bastará alterarmos as constantes.

```
int ehdirecao(char direcao) {
    return
        direcao == ESQUERDA ||
        direcao == CIMA ||
        direcao == BAIXO ||
        direcao == DIREITA;
}
```

Generalize o que fizemos nesta seção. Perceba que, no fundo, toda vez que falamos de boas práticas de código, fazemos a mesma coisa: isolar trechos de

código para que possamos reutilizá-lo em diversos pontos da aplicação, de forma que fique fácil saber o que ele faz apenas lendo o seu nome.

Está perdido?

O código até o momento está em http://pastebin.com/BjScydHn

18.3 Usando estruturas auxiliares

O próximo passo é fazer os fantasmas andarem pelo mapa. Entretanto, ainda não definimos como representaremos um fantasma no mapa. Assim como nos outros casos, escolheremos uma caractere para tal: `F`.

Vamos colocar alguns fantasmas no mapa, para começarmos nossos testes:

```
5 10
|--------|
|.F.|....|
|..-|.@..|
|..F.....|
|--------|
```

Agora precisamos movimentá-los no mapa. O algoritmo pode ser o quão complicado quisermos: eles poderiam ter algum tipo de inteligência artificial, por exemplo. Mas começaremos com algo simples, fazendo todos eles andarem, por enquanto, somente para a direita.

Para tal, varreremos a matriz em busca dos fantasmas. Assim que os encontrarmos, faremos andar uma posição à direita cada um deles. A função `fantasmas()` será responsável por isso. Ao encontrarmos um fantasma, verificaremos se a próxima posição à direita é válida. Se sim, então o fantasma andará para a direita:

```
void fantasmas() {
    for(int i = 0; i < m.linhas; i++) {
        for(int j = 0; j < m.colunas; j++) {
```

```
            if(m.matriz[i][j] == FANTASMA) {
                if(ehvalida(&m, i, j+1)) {
                    andanomapa(&m, i, j, i, j+1);
                }
            }

        }
    }
}
```

Invocaremos a função `fantasmas()` logo após a função `move()`, dentro do nosso loop principal do jogo:

```
do {
    imprimemapa(&m);

    char comando;
    scanf(" %c", &comando);

    move(comando);
    fantasmas();

} while (!acabou());
```

Se rodarmos nosso programa do jeito que está, teremos uma surpresa: o fantasma andará, não só uma, mas todas as casas que ele puder para a direita. Entender por que isso aconteceu é importante, pois nos dará um entendimento ainda maior sobre algoritmos e lógica de programação.

Veja, por exemplo, o fantasma que está em `(3,3)` do mapa. Nosso algoritmo o encontrará quando `i` for igual a 3, e `j` for também igual a 3. A próxima posição, `(3,4)` é vazia, e portanto movemos o fantasma. Até então, tudo funcionando de acordo.

Entretanto, quando nosso loop iterar ou seja, `i` for igual a 3, e `j` igual a 4 , encontraremos o mesmo fantasma novamente. A posição à direita estará vazia, e o fantasma será movido para lá. O loop iterará novamente, `j` será igual a 5, e o mesmo fantasma estará lá novamente. Perceba que empurramos o mesmo fantasma várias vezes para a direita.

Precisamos encontrar uma maneira de saber se já empurramos aquele fantasma ou não. Podemos pensar em diversos algoritmos para isso, mas nossa solução será criar uma cópia do mapa. Com uma cópia, poderemos iterar em uma, e mover na outra. Assim, não encontraremos o mesmo fantasma duas vezes.

Vamos criar, portanto, uma função que copia um mapa. Ela receberá dois mapas: o destino e a origem. Copiar um mapa não é uma tarefa simples. Precisamos copiar os dois inteiros, `linhas` e `colunas`, alocar a matriz, e copiar linha a linha da matriz origem.

Para copiar uma linha, usaremos a função `strcpy()`. O nome já diz o que ela faz: copia uma string em outra. A função recebe dois parâmetros, "destino" e "origem". Essa função nada mais faz do que varrer o array e copiar o conteúdo na posição original na "destino", e parar quando encontrar um *null* (`).

Veja a função `copiamapa()`, dentro do `mapa.c`:

```
void copiamapa(MAPA* destino, MAPA* origem) {
    destino->linhas = origem->linhas;
    destino->colunas = origem->colunas;
    alocamapa(destino);
    for(int i = 0; i < origem->linhas; i++) {
        strcpy(destino->matriz[i], origem->matriz[i]);
    }
}
```

Ao começar, a função `fantasmas()` agora fará uma cópia do mapa `m` e, depois, varrerá cada caractere do mapa `copia`. Sempre que achar um fantasma, sabemos que ele também existe em `m`. Se for possível movê-lo ou seja, `ehvalida()` e `ehvazia()` retorna verdadeiro para `(i, j+1)` no mapa `m`, fazemos somente no mapa `m`.

Em seguida, como o fantasma não existirá na próxima posição à direita da `copia`, uma vez que esse mapa está intocado, não teremos mais o problema de o fantasma aparecer novamente na próxima iteração do loop.

Ao final, liberamos o mapa `copia`, pois fizemos uso de alocação dinâmica para criá-lo, e ele não é mais necessário.

Em código:

```
void fantasmas() {
    MAPA copia;

    copiamapa(&copia, &m);

    for(int i = 0; i < copia.linhas; i++) {
        for(int j = 0; j < copia.colunas; j++) {
            if(copia.matriz[i][j] == FANTASMA) {
                if(ehvalida(&m, i, j+1) && ehvazia(&m, i, j+1)) {
                    andanomapa(&m, i, j, i, j+1);
                }
            }
        }
    }

    liberamapa(&copia);
}
```

É comum fazermos uso de estruturas auxiliares para nossos algoritmos. Nesse caso em particular, criamos uma cópia do mapa para conseguirmos ter os valores originais mesmo após modificá-los.

Uma grande parte da vida do programador é criar **estruturas de dados** que facilitem a manipulação dos dados que temos. Esse é, aliás, um outro assunto para você estudar futuramente.

Outras funções de manipulação

Usamos a função `strcpy()` para copiar um array para o outro. Essa função, como já explicamos, serve para copiar strings. Mas, às vezes, queremos copiar outros elementos que não strings. Por exemplo, podemos querer uma `struct` inteira.

Imagine o seguinte código:

```
struct contato {
    char nome[20];
    char email[50];
};
```

```
typedef struct contato CONTATO;

int main() {
    CONTATO mauricio;
    strcpy(mauricio.nome, "Mauricio Aniche");
    strcpy(mauricio.email, "mauricio.aniche@caelum.com.br");
}
```

Se quiséssemos copiar a `struct` inteira em vez de copiar atributo por atributo, poderíamos fazer uso da instrução `memcpy()`. Como o nome já diz, ela copia o que está na memória. A assinatura dela é bem parecida com a da `strcpy()`, com a diferença de que, além de passar o destino e a origem, precisamos passar também a quantidade de bytes que queremos copiar.

No código a seguir, copiamos o contato `mauricio` para o contato `aniche`. Repare que usamos `sizeof()` para calcular a quantidade de bytes da `struct`:

```
CONTATO aniche;

memcpy(&aniche, &mauricio, sizeof(CONTATO));
```

Outra função bastante comum quando manipulamos a memória é a `memset()`. Geralmente a usamos para inicializar structs ou arrays com algum valor padrão (costuma ser "nulo"). Isso é especialmente útil, pois muitos compiladores de C não limpam as regiões de memória antes de liberá-la para o programador. Então é comum termos "lixo".

Para não termos nunca o problema de ter dados sujos, podemos sempre limpar nossas structs, por exemplo:

```
CONTATO marcelo;

// 0 significa NULO
memset(&marcelo, 0, sizeof(CONTATO));
```

ESTÁ PERDIDO?

O código feito até o momento está em http://pastebin.com/fwpRCbiD

É possível deixar esse código ainda mais legível, criando uma função para encapsular o teste do `if`. Veja que o condicional retorna verdadeiro se `ehvalida()` e `ehvazia()` retornam verdadeiros. Repare que ambas recebem inclusive os mesmos parâmetros. Podemos criar uma função que invoca essas duas funções e com um nome que explique melhor o que esse condicional significa.

```
int podeandar(MAPA* m, int x, int y) {
    return
        ehvalida(m, x, y) &&
        ehvazia(m, x, y);
}

// ...

if(podeandar(&m, i, j+1)) {
    andanomapa(&m, i, j, i, j+1);
}
```

Dessa forma, o condicional ficará mais claro, e nosso código ainda mais fácil de ser reutilizado.

18.4 Um pouco de inteligência artificial

Nosso fantasma ainda não é muito inteligente: ele só sabe andar para a direita. O problema é que, quando ele anda até a borda do mapa, ele fica parado lá para sempre. Vamos dar um pouco mais de inteligência a ele. O fantasma, agora, se não conseguir andar para a direita, tentará andar para alguma outra direção. A chance de ter uma direção válida é grande.

O que queremos fazer aqui é dar um pouco de **inteligência artificial**, ou seja, ensinar a máquina a tentar tomar a decisão que maximizará seu ganho. Em outras palavras, o fantasma agora não ficará parado, ele andará sempre que possível em alguma direção. Claro que esse é o exemplo mais simples possível de inteligência artificial. Essa é uma área gigante de estudo e requer dedicação. Porém, vamos devagar.

Faremos isso de maneira aleatória. Escolheremos, de maneira randômica, uma das 4 direções válidas. Se ela for válida, é para lá que o fantasma vai;

se não for, tentaremos novamente. Para que o programa não entre em um loop infinito, faremos isso no máximo 10 vezes. Se depois da décima tentativa ainda não tivermos encontrado uma possível direção, esse fantasma ficará parado nessa rodada.

Vamos começar definindo uma estrutura auxiliar que guardará as 4 possíveis direções. Veja que isso nos ajudará a não fazer 4 diferentes `ifs`: *if sorteado é direita, tente ir pra lá*, *if sorteado é esquerda, tente ir para lá*. E se andássemos em todas as direções possíveis, contando diagonais? Teríamos muito código. Pense sempre nisso. Muito código repetido, mesmo que seja um pouco diferentes um do outro, é sempre problemático. Procure criar estruturas que flexibilizem e facilitem seu código.

Começaremos por declarar uma matriz dentro da nova função `praondevaifantasma()`, e já diretamente inicializá-la com as 4 direções possíveis: `x, y+1` (direita), `x+1, y` (baixo), `x, y-1` (esquerda) e `x-1, y` (cima).

```
int praondefantasmavai(int xatual, int yatual,
    int* xdestino, int* ydestino) {

    int opcoes[4][2] = {
        { xatual   , yatual+1 },
        { xatual+1 , yatual   },
        { xatual   , yatual-1 },
        { xatual-1 , yatual   }
    };

}
```

Em seguida, criamos nosso loop de, no máximo, 10 iterações, dentro do qual sorteamos um número de 0 a 3 (a quantidade de linhas da matriz) e testamos essa direção. Se ela for válida, devolvemos a direção repare que aqui precisamos receber ponteiros, pois queremos que a função devolva dois números , e fazemos a função retornar verdadeiro. Caso contrário, se as 10 tentativas falharem, retornamos falso. Não podemos esquecer de incluir `time.h` no arquivo:

```
int praondefantasmavai(int xatual, int yatual,
    int* xdestino, int* ydestino) {
```

```
    int opcoes[4][2] = {
        { xatual   , yatual+1 },
        { xatual+1 , yatual   },
        { xatual   , yatual-1 },
        { xatual-1 , yatual   }
    };

    srand(time(0));
    for(int i = 0; i < 10; i++) {
        int posicao = rand() % 4;

        if(podeandar(&m, opcoes[posicao][0],
                    opcoes[posicao][1])) {
            *xdestino = opcoes[posicao][0];
            *ydestino = opcoes[posicao][1];
            return 1;
        }
    }

    return 0;
}
```

Agora, basta mudarmos o algoritmo da função `fantasmas()`. Ao encontrarmos um fantasma no mapa, invocaremos a função `praondefantasmavai()` para descobrir o caminho dele. Se a função nos retornar verdadeiro, sabemos que podemos movê-lo para aquela direção:

```
void fantasmas() {
    MAPA copia;

    copiamapa(&copia, &m);

    for(int i = 0; i < copia.linhas; i++) {
        for(int j = 0; j < copia.colunas; j++) {
            if(copia.matriz[i][j] == FANTASMA) {

                int xdestino;
```

```
                int ydestino;

                int encontrou = praondefantasmavai(i, j,
                                                   &xdestino,
                                                   &ydestino);

                if(encontrou) {
                    andanomapa(&m, i, j, xdestino, ydestino);
                }
            }
        }
    }

    liberamapa(&copia);
}
```

Pronto. Nossos fantasmas agora andam de maneira aleatória pelo mapa e têm um pouco de inteligência na hora de andar. É claro que poderíamos melhorar ainda mais, por exemplo, fazendo os fantasmas sempre andarem na direção do herói, ou mesmo trabalhar em equipe, tentando cercá-lo. Esse é um bom desafio para você.

Está perdido?

O código até o momento está em http://pastebin.com/knjL0xYH

18.5 Acoplamento, encapsulamento e assinaturas de funções

O que falta agora é fazermos a finalização do jogo. O Foge-foge é um jogo no qual não há vitória. Ele só acaba quando o herói morre. Em nosso jogo, o herói morrerá quando o fantasma "passar por cima dele".

Descobrir isso é fácil. Como sabemos que os fantasmas andam depois do personagem, se algum fantasma passar por cima do herói, então, ao final do turno, não haverá herói no mapa. Já temos uma função que nos ajuda a

encontrar algum personagem no mapa, a `encontramapa()`. No entanto, precisaremos adaptá-la para nos avisar caso o personagem não seja encontrado.

Mudaremos, então, a assinatura da função, fazendo-a devolver agora um inteiro, que trataremos como booleano. Ela nos devolverá verdadeiro, caso encontremos o personagem, e falso, caso contrário:

```
int encontramapa(MAPA* m, POSICAO* p, char c) {

    for(int i = 0; i < m->linhas; i++) {
        for(int j = 0; j < m->colunas; j++) {
            if(m->matriz[i][j] == c) {
                p->x = i;
                p->y = j;
                return 1;
            }
        }
    }

    // não encontramos!
    return 0;
}
```

Repare que nosso código continua compilando, mesmo após termos mudado a função. Isso porque fizemos a mudança mais simples que podíamos na assinatura dela: trocamos de `void` para `int`.

Mas o que aconteceria se trocássemos, por exemplo, o nome ou mesmo os parâmetros que ela recebe? Nosso código pararia de compilar. Todo lugar que faz uso dela deixaria de funcionar, uma vez que a função mudou.

Não é fácil mudar a assinatura de uma função. Perceba o **acoplamento** entre a função e os diversos lugares que fazem uso dela. Pensar muito bem na sua assinatura é importantíssimo. Nossos programas, até então, são pequenos, mas mudar pode ser complicado em projetos maiores.

Parte do trabalho do bom desenvolver é justamente pensar nos "contratos" que as funções têm. Por contrato, queremos dizer o nome da função pois é por meio dele que você sabe o que a função faz , os parâmetros que ela re-

cebe e o que ela devolve. Se o contrato estiver bem definido, você raramente precisará mudar a assinatura.

A vantagem de ter pensado bem no contrato é que a implementação depois pode variar. Você reparou que, em vários momentos do livro, mudamos o conteúdo da função? É o caso da função `fantasma()`. Alteramos o comportamento do andar dos fantasmas ao longo deste capítulo. Ou seja, mudamos nosso algoritmo para algo melhor, mas sem "quebrar" o resto. Tudo continuou compilando.

Quando você estudar Orientação a Objetos, daremos a isso o nome de **encapsulamento**. Encapsular significa esconder os detalhes do algoritmo dentro da função. Isso é feito para que possamos mudá-los depois, sem nos preocuparmos em fazer o resto do sistema parar, disponibilizando para aqueles que querem fazer uso dela, um bom contrato que, por ser bom, tenha baixa chance de mudança.

Como lá, no começo do nosso jogo, pensamos bem que precisaríamos de uma função `acabou()`, agora basta preenchê-la com um algoritmo de verdade para decidir se o jogo acabou ou não. O que faremos é declarar uma estrutura `POSICAO` "inútil" e invocar a função `encontramapa()`, que procurará pelo herói. Se ela retornar 0, então a função `acabou()` retornará o oposto disso. Ou seja, se não encontrarmos o herói, `acabou()` retorna 1:

```
int acabou() {
    POSICAO pos;
    return !encontramapa(&m, &pos, HEROI);
}
```

Se jogarmos agora, veremos que estamos quase lá, mas ainda não está certo. Os fantasmas não conseguem passar por cima do herói. Além disso, o herói não consegue comer fantasmas. Precisamos melhorar nosso algoritmo.

O problema está na função `podeandar()`. Repare que o algoritmo lá era bem simplista. Ele permitia o personagem andar se a casa à direita estivesse vazia ou fosse vazia:

```
int podeandar(MAPA* m, int x, int y) {
    return
        ehvalida(m, x, y) &&
```

```
        ehvazia(m, x, y);
}
```

Essa função precisa ser mais inteligente. Podemos andar quando:

I) A próxima posição estiver dentro dos limites do mapa (é exatamente o que a função `ehvalida()` faz).

II) A próxima posição não for uma parede (precisamos criar essa função).

III) A próxima posição não for um personagem idêntico ao que está andando (esse é o caso particular para que um fantasma não passe por cima de outro).

Vamos, então, criar a função `ehparede()` e deletar a função `ehvazia()`, pois não precisaremos mais dela. Acostume-se também com isso: se você não usa mais a função, apague-a. Código que não é usado só serve para complicar a vida do programador.

```
int ehparede(MAPA* m, int x, int y) {
    return
        m->matriz[x][y] == PAREDE_VERTICAL ||
        m->matriz[x][y] == PAREDE_HORIZONTAL;
}
```

Agora vamos criar uma função que compara o personagem que está na posição com o que foi passado por parâmetro. Apesar de essa função ter apenas uma linha, é muito melhor ter uma função dessa do que ter que ler o `if` e entender o que está acontecendo. Lembre-se de que o nome da função é importante, e nos ajuda a ler menos.

```
int ehpersonagem(MAPA* m, char personagem, int x, int y) {
    return
        m->matriz[x][y] == personagem;
}
```

Por fim, vamos mudar a função `podeandar()`. Dessa vez, não teremos jeito: precisaremos mudar a assinatura dela para receber o personagem que

está andando. Infelizmente, precisaremos procurar por todos os lugares que a invocam e passar o novo parâmetro. Paciência, não pensamos nisso antes.

Repare que conseguimos ler seu código do mesmo jeito que lemos a listagem anterior: *pode andar se é válida E se não é parede E se não é personagem*.

```
int podeandar(MAPA* m, char personagem, int x, int y) {
    return
        ehvalida(m, x, y) &&
        !ehparede(m, x, y) &&
        !ehpersonagem(m, personagem, x, y);
}
```

Como mudamos a assinatura da função, vamos mudar também nos lugares que a invocam. Por sorte, temos apenas dois lugares: dentro da função `move()` e dentro da função `praondefantasmavai()`. Basta apenas passarmos a constante do respectivo personagem:

```
int praondefantasmavai(int xatual, int yatual,
    int* xdestino, int* ydestino) {

    // ...

    if(podeandar(&m, FANTASMA,
        opcoes[posicao][0], opcoes[posicao][1])) {
        // ...
    }

    // ...
}

void move(char direcao) {
    // ...

    if(!podeandar(&m, HEROI, proximox, proximoy))
        return;

    // ...
}
```

Agora sim. Fantasmas podem passar por cima do herói, e o herói pode passar por cima dos fantasmas. Mas o que acontece se nosso herói passar por cima de todos os fantasmas? O jogo perde a graça. Vamos terminar o jogo também nesse caso. Isso é fácil, a função `acabou()` serve exatamente para isso, e a ideia é análoga à que já temos: se não houver mais fantasmas no mapa, o jogo também acabou.

Note aqui como criamos duas variáveis, `perdeu` e `ganhou`, apenas para aumentar a legibilidade do código. Essa é outra estratégia que você pode usar para facilitar a leitura: criar variáveis que explicam o retorno de uma função (ou uma conta, ou qualquer que seja o conteúdo que ela armazena):

```
int acabou() {
    POSICAO pos;

    int perdeu = !encontramapa(&m, &pos, HEROI);
    int ganhou = !encontramapa(&m, &pos, FANTASMA);

    return ganhou || perdeu;

}
```

Pronto. Os fantasmas agora andam de maneira aleatória e podem matar o herói. O jogo está ficando quente.

Está perdido?

O código até o momento está em http://pastebin.com/gYFjK07E

18.6 Resumindo

Neste capítulo, aprendemos:

- Novamente, a extrair pequenas funções que são reutilizáveis;
- Que podemos (e devemos) criar estruturas auxiliares para facilitar a criação dos nossos algoritmos;

- Que podemos não abrir chaves no `if` e que, nesse caso, a execução do condicional será apenas a próxima instrução que aparecer;
- Que podemos copiar strings usando `strcpy()`.
- Que podemos copiar ou limpar structs, arrays ou qualquer ponteiro, usando `memcpy()` e `memset()`;
- Que precisamos pensar bem no contrato das nossas funções, pois mudá-los é difícil;
- Que podemos usar variáveis apenas para facilitar a legibilidade da nossa função.

Capítulo 19

Recursividade

A próxima funcionalidade do nosso jogo é dar a possibilidade de nosso herói comer a pílula do super poder. Se ele capturar essa pílula, apertando a tecla B, ele soltará uma bomba que destruirá todos os fantasmas que estiverem a até 3 casas de distância, à direita.

Imagine o mapa a seguir, por exemplo, e suponha que W também seja fantasma. Todos os fantasmas anotados como W morrerão, caso o herói opte por soltar a bomba naquele momento. Perceba que todos eles estão no máximo a 3 casas de distância.

```
|--------|
|.....F..|
|.....-..|
|.@W.W...|
|.. ..FF.|
|.....F..|
```

```
|..F..F..|
|--------|
```

Vamos começar pela lógica de capturar a pílula. Ela será representada pela letra `P` no mapa. Caso o jogador mova o herói em uma casa que a tenha, devemos guardar isso de alguma forma: a variável global `tempilula` nos dirá quantas pílulas o jogador tem para gastar. Vamos também inicializá-la com 0, uma vez que o jogador começa sem nenhuma pílula:

```
MAPA m;
POSICAO heroi;

// nova variável global
int tempilula = 0;
```

Precisamos também criar a constante `PILULA`, adicioná-la ao header file do mapa, e mudar o mapa:

```
// mapa.h
#define PILULA 'P'

// mapa.txt
5 10
|--------|
|FFF|....|
|FF-|.@P.|
|..F.....|
|--------|
```

Agora, basta igualarmos essa variável a 1 sempre que o herói andar por cima de uma dessas pílulas. Para isso, na função `move()`, basta adicionarmos a verificação: a casa destino tem uma pílula? Se sim, coloque verdadeiro nela:

```
void move(char direcao) {
    // ...

    if(!podeandar(&m, HEROI, proximox, proximoy))
        return;
```

```
    if(ehpersonagem(&m, PILULA, proximox, proximoy)) {
        tempilula = 1;
    }

    // ...
}
```

Vamos até exibir para o jogador a quantidade de pílulas que ele tem naquele momento. Vamos colocar esse código dentro do método `main()`, embora não seja o local ideal mas depois discutimos sobre isso.

```
int main() {

    // ...

    do {
        printf("Pílula: %s\n", (tempilula ? "SIM" : "NÃO"));
        imprimemapa(&m);

        // ...
    }

    // ...

}
```

A próxima etapa é fazer nosso programa entender outras entradas que não uma direção. Ou seja, se a tecla pressionada for uma direção, o método `move()` deve ser invocado. Porém, se a tecla for um `B`, ele deve fazer a bomba explodir.

Precisamos fazer o teste da tecla digitada agora no loop principal do programa, e não mais dentro da função `move()`. Dessa forma, ele conseguirá dizer qual função deverá ser invocada. A constante `BOMBA` deve ser definida no header file:

```
// fogefoge.h
#define BOMBA 'b'

// fogefoge.c
```

```
do {
    printf("Pílula: %s\n", (tempilula ? "SIM" : "NÃO"));
    imprimemapa(&m);

    char comando;
    scanf(" %c", &comando);

    if(ehdirecao(comando)) move(comando);
    if(comando == BOMBA) explodepilula();

    fantasmas();

} while (!acabou());
```

Vamos declarar a função `explodepilula()` apenas para nosso código compilar no momento, mas ainda sem uma implementação concreta, pois ela será o alvo da discussão da próxima seção:

```
void explodepilula() {

}
```

Por que a main não é o melhor lugar?

Lembre-se do encapsulamento e de que você deve fazer o programador ler a menor quantidade de código possível.

Idealmente, a função `main()` apenas invoca e coordena o programa como um todo. Toda e qualquer regra de negócio deve estar encapsulada em uma função com um bom contrato. Ou seja, esse `printf()` mais o `imprimemapa()` deveriam estar encapsulados em uma função.

Está perdido?

O código até o momento está em http://pastebin.com/L27AnPjN

19.1 Entendendo recursão

Implementar a destruição dos fantasmas à direita é razoavelmente fácil. Basta fazermos um loop contado 3 vezes, e verificar se a posição `x, y+i` (onde `x` e `y` vem da posição `heroi`) é válida e não é uma parede. Caso isso seja verdade, basta trocarmos aquela posição por vazio:

```
void explodepilula() {

    for(int i = 1; i <= 3; i++) {
        if(ehvalida(&m, heroi.x, heroi.y+i) &&
            !ehparede(&m, heroi.x, heroi.y+i)) {
            m.matriz[heroi.x][heroi.y+i] = VAZIO;
        }
    }

}
```

Podemos ainda tratar o caso da parede. Se tiver uma parede no caminho, o efeito da bomba para de propagar para as outras casas. Ou seja, a parede é imune à bomba. Para isso, mudaremos um pouco a maneira como nosso algoritmo foi desenhado. Assim que encontrarmos uma parede, quebramos o loop.

```
void explodepilula() {

    for(int i = 1; i <= 3; i++) {
        if(ehvalida(&m, heroi.x, heroi.y+i)) {
            if(ehparede(&m, heroi.x, heroi.y+1)) break;

            m.matriz[heroi.x][heroi.y+i] = VAZIO;
        }
    }

}
```

Está perdido?

O código feito até o momento está em http://pastebin.com/Fqq7DWGB

Ótimo. Temos nosso algoritmo funcionando para uma direção. Mas, como sabemos, temos várias maneiras diferentes de implementar o mesmo algoritmo. Vamos tentar algo diferente agora. Sabemos que a principal parte dessa função é justamente a linha em que colocamos `VAZIO` na posição:

```
m.matriz[x][y] = VAZIO;
```

Vamos colocar somente isso na nossa função, que receberá `x` e `y` como parâmetros:

```
void explodepilula(int x, int y) {

    m.matriz[x][y] = VAZIO;

}
```

Sabemos também que depois de marcar a posição como `VAZIO`, o mesmo deve acontecer para a posição `y+1`. É o que faremos: **chamaremos a função dentro dela mesma**, passando `y+1` dessa vez. Sim, isso é possível, e chamamos isso de **função recursiva**. Ou seja, ela invoca a si mesma:

```
void explodepilula(int x, int y) {

    m.matriz[x][y+1] = VAZIO;
    explodepilula(x, y+1);
}
```

Como mudamos a sua assinatura, precisamos mudar sua invocação também:

```
if(comando == BOMBA) explodepilula(heroi.x, heroi.y);
```

Pare e pense no que está acontecendo. Invocamos a função `explodepilula()` pela primeira vez dentro do método `main`. Lá, passamos `heroi.x` e `heroi.y` como parâmetros. Suponha que eles valham `1,1` nesse momento. A função começa e limpa o que está na posição `1,2` (repare o +1 dentro da matriz). Em seguida, ela invoca a si mesma, com os parâmetros `x, y+1`, ou seja, `1, 2`. Vamos para a segunda execução da função. Aqui, ela limpa o que está na posição, e se autoinvoca novamente, dessa vez, com os valores `1, 3`. E assim por diante.

Você talvez tenha visto um problema nisso. Se não conseguiu perceber, rode a aplicação e exploda a bomba. Você verá um erro:

```
Segmentation fault: 11
```

Você percebeu que a função nunca parou de invocar a si mesma? Ela se chama para `1,1`, depois para `1,2`, então para `1,3`, e repete isso até o infinito. Ou seja, temos uma chamada recursiva infinita. A função fica se invocando até que não haja mais espaço na máquina para guardar a pilha de execução. E aí, temos o tal do **segmentation fault**.

Quando criamos funções recursivas, precisamos deixar claro em que momento ela parará de chamar a si mesma. Isso depende do problema que estamos resolvendo. Aqui, sabemos que precisamos executar a função 3 vezes. Logo, colocaremos um "contador". Ou seja, uma variável `qtd` (de quantidade), que quando for o significa que a função deve parar de chamar a si própria. E como fazemos esse número ser zero? Na invocação recursiva, passamos `qtd-1`. Em código:

```
void explodepilula(int x, int y, int qtd) {
    // se acabou o número de vezes,
    // então acaba a função
    if(qtd == 0) return;

    m.matriz[x][y+1] = VAZIO;

    // dessa vez, passamos qtd-1, pois
    // já rodamos uma vez a função
    explodepilula(x, y+1, qtd-1);
}
```

Mudamos também a invocação no método `main`:

```
// 3 é a quantidade de casas que temos que explodir
if(comando == BOMBA) explodepilula(heroi.x, heroi.y, 3);
```

Vamos agora simular a execução dessa função. A função `explodepilula()` é invocada na `main()`. Vamos supor que os parâmetros passados para ela sejam `1, 1, 3`. Na primeira execução, a função limpa a posição `1, 2` e autoinvoca-se novamente com os parâmetros `1, 2, 2`. A primeira execução é, portanto, suspensa para que a segunda aconteça. Nela, a posição `1, 3` é limpa, e ela autoinvoca-se com os valores `1, 3, 1`. A segunda invocação, então, é suspensa para a terceira execução. A casa `1, 4` é limpa, e a função é novamente invocada com os valores `1, 4, 0`.

Como `qtd` é igual a zero, a quarta execução morre. Com isso, a terceira execução volta à ativa e acaba, já que não há mais nada nela. A segunda invocação volta e também acaba. O mesmo acontece com a primeira, finalizando a execução da função como um todo.

Acabamos de escrever nossa primeira função recursiva. Uma função recursiva é aquela que invoca a si mesma ao longo de sua execução. Para que isso funcione corretamente, ela precisa em algum momento parar de invocar-se (chamamos isso de **ponto de fuga**).

Fibonacci

Podemos escrever muitas outras funções de maneira recursiva. A função matemática de Fibonacci é um bom exemplo. Ela é expressa por meio da seguinte fórmula:

```
F(0) = 0
F(1) = 1
F(n) = F(n-1) + F(n-2)
```

Pela natureza da função, perceba que ela invoca a ela mesma (ou seja, recursiva). Fibonacci de um número qualquer é igual à soma do Fibonacci do número anterior mais o Fibonnaci do número anterior ao anterior.

Podemos escrever a seguinte função recursiva para calculá-lo. Repare que sempre precisamos de um ponto de fuga e, nesse caso, a temos. Afinal, sabemos que Fibonacci de zero é 0, e Fibonacci de 1 é 1.

Em código, teríamos algo como:

```
int fib(int n) {
    if(n == 0) return 0;
    if(n == 1) return 1;

    return fib(n-1) + fib(n-2);
}
```

Note que a função invoca a si mesma passando sempre -1 e -2. Em algum momento, esses parâmetros serão iguais a 0 ou 1, e aí a função entra em seu ponto de fuga e para de invocar a si mesma.

Um bom exercício é desenhar a árvore de invocações e entender bem o que acontece quando fazemos `fib(5)`, por exemplo.

Está perdido?

O código até o momento está em http://pastebin.com/gsEqr1rD

19.2 Complicando o algoritmo recursivo

Precisamos melhorar essa função. Afinal, ela ainda ignora possíveis exceções, como o caso de uma das 3 casas à direita não serem válidas ou serem paredes, por exemplo. Esses são somente outros pontos de fuga da função. Caso alguma dessas afirmações seja verdade, terminamos a sua execução. Em código:

```
void explodepilula(int x, int y, int qtd) {

    if(qtd == 0) return;
    if(!ehvalida(&m, x, y+1)) return;
    if(ehparede(&m, x, y+1)) return;

    m.matriz[x][y+1] = VAZIO;
    explodepilula(x, y+1, qtd-1);
}
```

Agora, precisamos expandir a ideia para as quatro possíveis direções, e não só para a direita. Repare que a única coisa que muda de uma direção para a outra são as contas que fazemos com `x` e `y`. Para a direita, somamos `0,1`, já para a esquerda, somaríamos `0,-1`. Para cima, `-1, 0`, e para baixo, `1, 0`.

Vamos receber esses números como parâmetros da função. Dessa forma, a função fica genérica o suficiente para servir a todos os casos. Dessa vez, faremos uso de variáveis auxiliares `novox` e `novoy`. Vamos, inclusive, renomear essa função para `explodepilula2`:

```
void explodepilula2(int x, int y, int somax, int somay, int qtd){

    if(qtd == 0) return;
```

```
    int novox = x+somax;
    int novoy = y+somay;

    if(!ehvalida(&m, novox, novoy)) return;
    if(ehparede(&m, novox, novoy)) return;

    m.matriz[novox][novoy] = VAZIO;
    explodepilula2(novox, novoy, somax, somay, qtd-1);
}
```

Renomeamos a função justamente para criar a função `explodepilula()`, que fará as 4 invocações necessárias para cada direção. É essa a função que será invocada dentro da `main()`:

```
void explodepilula() {
    explodepilula2(heroi.x, heroi.y, 0, 1, 3);
    explodepilula2(heroi.x, heroi.y, 0, -1, 3);
    explodepilula2(heroi.x, heroi.y, 1, 0, 3);
    explodepilula2(heroi.x, heroi.y, -1, 0, 3);
}
```

Funções recursivas têm vantagens. Elas geralmente são mais simples e fáceis de serem lidas e implementadas. Mas, dependendo do caso, elas podem não ser tão performáticas. Afinal, sempre que invocamos uma função dentro da outra, o programa é obrigado a manter todas as variáveis que existem na primeira função em algum canto da memória, até que a segunda função invocada acabe e a primeira volte a ser executada.

Se tivermos, por exemplo, 1000 funções invocadas na sequência, a máquina é obrigada a guardar as variáveis das 999 chamadas anteriores, que ainda não acabaram. O mesmo não aconteceria se tivéssemos usado algum tipo de loop, por exemplo.

Podemos prolongar a discussão aqui. Muitos compiladores modernos conseguem reescrever a recursão no momento da compilação e transformá-la em um loop convencional, que ocupa menos espaço da pilha da execução. Mas esse não é o foco. O objetivo deste capítulo é fazer você entender o que são funções recursivas e como escrevê-las.

Para terminar, falta fazer a função funcionar somente se temos bombas e retirar a bomba caso a usemos. Podemos fazer isso facilmente dentro da função `explodepilula()`. Basta colocar `0` dentro da variável `tempilula`:

```
void explodepilula() {

    // se não tem pílula, não pode fazer nada
    if(!tempilula) return;

    explodepilula2(heroi.x, heroi.y, 0, 1, 3);
    explodepilula2(heroi.x, heroi.y, 0, -1, 3);
    explodepilula2(heroi.x, heroi.y, 1, 0, 3);
    explodepilula2(heroi.x, heroi.y, -1, 0, 3);

    // tira a pílula que acabou de usar
    tempilula = 0;
}
```

Está perdido?

O código até o momento está em http://pastebin.com/H9PcJcHa

19.3 Resumindo

Neste capítulo, aprendemos:

- O que são funções recursivas;
- Que funções recursivas precisam de pontos de fuga, ou seja, pontos onde elas não invocam mais a si próprias.

Capítulo 20

Outras diretivas de compilação

O último passo agora é deixar nosso jogo bonito, já que ele não está nada apresentável. Como já sabemos que precisamos separar as responsabilidades ao máximo, vamos mover a função `imprimemapa()` para um novo arquivo, o `ui.c` (UI vem de *User Interface*, ou interface do usuário).

O primeiro passo é definirmos como imprimiremos cada personagem do mapa. Se abusarmos um pouco novamente de ASCII Art, podemos desenhar um fantasma de maneira muito mais elegante. Mas precisaremos de mais de uma linha para isso. Vamos declarar uma variável que guarda o desenho do nosso fantasma, em várias linhas. O desenho terá 4 linhas por 6 caracteres. Repare, no entanto, que declaramos 7 posições na segunda dimensão da matriz, justamente para guardar o ` final:

```
char desenhofantasma[4][7] = {
    {" .-.  " },
```

```
    {"| OO| " },
    {"|   | " },
    {"'^^^' " }
};
```

A ideia é que, quando encontrarmos um `F` no mapa, imprimiremos esse desenho. Quando encontrarmos um `@`, imprimiremos o herói, e assim por diante. Vamos criar, portanto, o desenho dos outros possíveis caracteres do nosso mapa:

```
char desenhoparede[4][7] = {
    {"......" },
    {"......" },
    {"......" },
    {"......" }
};


char desenhoheroi[4][7] = {
    {" .--. "  },
    {"/ _.-'"  },
    {"\\  '-." },
    {" '--' "  }
};

char desenhopilula[4][7] = {
    {"      "},
    {" .-.  "},
    {" '-'  "},
    {"      "}
};

char desenhovazio[4][7] = {
    {"      "},
    {"      "},
    {"      "},
    {"      "}
};
```

Com isso em mãos, precisamos agora imprimir os desenhos na hora certa.

Mas, diferentemente do original, onde fazíamos dois loops aninhados (um para cada dimensão da matriz), aqui precisaremos ser um pouco mais inteligentes.

Note que cada desenho tem 4 linhas. Como estamos usando a API simples de escrever na tela, não conseguimos escrever 4 linhas para desenhar o primeiro caractere que aparece no mapa, e depois voltar para a primeira linha novamente, para começar a desenhar o segundo.

Precisamos ler a primeira linha do mapa e imprimir as primeiras linhas de todos os personagens que estão ali. Depois, todas as segundas linhas, todas as terceiras e, por fim, todas as quartas linhas. Só, então, poderemos começar a impressão da segunda linha do mapa.

Para isso, precisamos de mais um loop contado de 1 até 4 (ou 0 a 3), que imprimirá a linha certa de todos os desenhos. Veja o código a seguir. Temos a linha em `i`, a coluna em `j` e a parte do desenho a ser impressa em `parte`:

```
void imprimemapa(MAPA* m) {

    for(int i = 0; i < m->linhas; i++) {
        for(int parte = 0; parte < 4; parte++) {
            for(int j = 0; j < m->colunas; j++) {

                // é aqui que imprimiremos

            }

        }
}
```

Vamos criar uma função que apenas imprimirá a parte especificada do desenho. A assinatura dessa função deve ser `imprimeparte(char desenho[4][7], int parte)`, pois precisamos receber a matriz de `4x7` e o inteiro que nos diz a parte:

```
void imprimeparte(char desenho[4][7], int parte) {
    printf("%s", desenho[parte]);
}
```

Agora, basta fazermos a função `imprimemapa()` invocar a função `imprimeparte()` de acordo com o caractere que estiver em `m->matriz[i][j]` e o número da parte a ser impressa. Um simples `switch` resolve o problema.

Repare na sintaxe que usamos em `PAREDE_VERTICAL` e `PAREDE_HORIZONTAL`. Como em ambos os casos queremos imprimir o desenho `desenhoparede`, podemos colocar um `case` embaixo do outro e, dessa forma, ambos terão o mesmo comportamento:

```
void imprimemapa(MAPA* m) {
    for(int i = 0; i < m->linhas; i++) {

        for(int parte = 0; parte < 4; parte++) {
            for(int j = 0; j < m->colunas; j++) {

                switch(m->matriz[i][j]) {
                    case FANTASMA:
                        imprimeparte(desenhofantasma, parte);
                        break;
                    case HEROI:
                        imprimeparte(desenhoheroi, parte);
                        break;
                    case PILULA:
                        imprimeparte(desenhopilula, parte);
                        break;
                    case PAREDE_VERTICAL:
                    case PAREDE_HORIZONTAL:
                        imprimeparte(desenhoparede, parte);
                        break;
                    case VAZIO:
                        imprimeparte(desenhovazio, parte);
                        break;
                }

            }
            printf("\n");
        }
```

```
    }
}
```

No `ui.c`, não podemos esquecer de importar os header files necessários:

```
#include <stdio.h>
#include "mapa.h"
```

Com o algoritmo pronto, precisamos apenas acertar as arestas. Precisamos garantir que a função `imprimemapa()` não exista mais nem em `mapa.c` e nem em `mapa.h`. Ela agora está em `ui.h`. Observe que ele precisa incluir `mapa.h`, uma vez que ele faz uso de `MAPA` por lá:

```
#include "mapa.h"

void imprimeparte(char desenho[4][7], int parte);
void imprimemapa(MAPA* m);
```

Precisamos incluir `ui.h` dentro de `fogefoge.c`; afinal, é ele que invoca (já invocava, aliás) a função `imprimemapa()`.

```
#include <stdio.h>
#include <stdlib.h>
#include "time.h"
#include "fogefoge.h"
#include "mapa.h"

// novo include
#include "ui.h"
```

O código está todo certo. Porém, ao tentarmos compilar nossos 3 arquivos juntos (sim, agora são 3!), recebemos diversos erros parecidos com os que seguem:

```
> gcc fogefoge.c mapa.c ui.c -o fogefoge.out

In file included from fogefoge.c:6:
In file included from ./ui.h:4:
./mapa.h:12:8: error: redefinition of 'mapa'
struct mapa {
```

```
       ^
./mapa.h:12:8: note: previous definition is here
struct mapa {
       ^
```

O erro nos diz que estamos *redefinindo mapa*. Ou seja, por algum motivo ele encontrou uma nova declaração da `struct mapa` em nosso código. Mas, se temos certeza de que isso não acontece, por que o compilador se perdeu?

20.1 Ifdefs e Ifndefs

O problema acontece porque, quando fazemos um *include*, o compilador literalmente inclui o código daquele arquivo ali. É como se fosse um *copia e cola* feito de forma automática pelo compilador. E como agora temos uma mistura de inclusões, já que o `fogefoge.c` inclui tanto `mapa.h` quanto `ui.h`, e o `ui.h` também inclui `mapa.h`, acabamos por repetir a mesma declaração.

A primeira solução para isso é achar a combinação de inclusões a se fazer. Por exemplo, se sabemos que o `ui.h` inclui `mapa.h`, só precisamos incluir o primeiro, pois o segundo já será incluído por ele. O problema é que essa solução pode ficar bastante complicada à medida que nosso programa cresce e torna-se mais complicado.

Precisamos de uma maneira de dizer ao compilador para não incluir um arquivo se ele já foi incluído. Ou para ignorar uma segunda declaração.

Similar às diretivas `include` e `define`, o compilador da linguagem C nos dá outros 2 interessantes: o `ifdef` e o `ifndef`. Como o próprio nome diz, o que eles fazem são `ifs`. Mas são `ifs` diferentes: eles acontecem em tempo de compilação. O restante do nome, o `def`, é porque o condicional é feito em cima de um `define`.

Entender isso parece complicado, porém, em código, é muito mais simples. Veja o código a seguir. Temos 3 `printfs`. No entanto, o segundo está entre um `ifdef`. Repare que a condição é se o `#define IMPRIME` existe. Como ele não existe, a saída desse programa é simplesmente: `antes do imprimiu` e `depois do imprimiu`.

```
#include <stdio.h>
```

```
int main() {

    printf("antes do imprimiu\n");

    // esse trecho de código não será compilado,
    // pois o compilador viu que a constante
    // IMPRIME nunca foi definida
    #ifdef IMPRIME
        printf("imprimiu\n");
    #endif
    printf("depois do imprimiu");

}
```

No entanto, se definirmos uma constante `IMPRIME` com qualquer valor (ou até mesmo vazio), no momento da compilação, o compilador perceberá que aquela constante existe e compilará o que está dentro do `ifdef`:

```
// definimos a constante e,
// mesmo que vazia, ela existe!
#define IMPRIME

int main() {

    printf("antes do imprimiu\n");

    // o compilador percebe que ela existe,
    // então compilará o que está dentro do ifdef.
    #ifdef IMPRIME
        printf("imprimiu\n");
    #endif
    printf("depois do imprimiu");

}
```

A grande charada aqui é perceber que esse teste é feito em tempo de compilação. E, se o teste falhar, o compilador nem olhará para o que estiver dentro do `ifdef`: ele realmente ignorará e não compilará o que estiver lá. Claramente, o `ifndef` é idêntico: a diferença é que ele verifica se a constante não

foi definida.

Sabendo disso, a solução elegante para o problema das redefinições e includes por todo lado é usar essas diretivas de compilação dentro de nossos .h. A primeira coisa que faremos em cada um deles é perguntar se uma constante qualquer por exemplo, _UI_H_ existe. Se ela não existir, significará para nós que aquele header não foi incluído por nenhum outro arquivo. Logo, dentro do ifndef, colocaremos todo o conteúdo que queremos, como declaração de funções e structs, e definiremos a constante _UI_H_. A partir de agora, ela existe. Isso significa que, na próxima vez que incluirmos esse header, a constante existirá, o ifndef será falso, e o compilador não passará novamente pelas mesmas definições de funções e struct, evitando o problema da redeclaração.

Em código, o ui.h ficaria:

```
#ifndef _UI_H_
#define _UI_H_

#include "mapa.h"

void imprimeparte(char desenho[4][7], int parte);
void imprimemapa(MAPA* m);

#endif
```

O mesmo aconteceria com o mapa.h. Obviamente, daremos um nome diferente à constante que é usada:

```
#ifndef _MAPA_H_
#define _MAPA_H_

// todo o conteúdo aqui dentro

#endif
```

E fogefoge.h:

```
#ifndef _FOGEFOGE_H_
#define _FOGEFOGE_H_
```

```
// todo conteúdo aqui dentro

#endif
```

Perceba que o compilador é poderoso. Podemos até dizer a ele o que deve ou não ser compilado. Na prática, usamos muito para evitar o problema da redeclaração de coisas que estão dentro de nossos header files. Assim, podemos incluir o mesmo arquivo várias vezes, sem nos preocuparmos com detalhes.

Outro cenário onde diretivas de compilação são muito utilizadas é o caso de desenvolvimento de aplicativos para arquiteturas, plataformas ou sistemas operacionais diferentes. Todo o código que escrevemos neste livro é `ANSI C`, ou seja, é padrão. Ele pode ser compilado em qualquer sistema operacional como Windows, Linux e Mac , e o comportamento será o mesmo.

No entanto, temos muitas bibliotecas específicas de cada sistema operacional. Por exemplo, bibliotecas gráficas que só rodam no Windows, ou outras que só rodam no Linux. Nesses casos, muitos desenvolvedores optam por fazer grande parte do código em `ANSI C` e, nos pequenos pedaços específicos de cada sistema operacional, usar diretivas de compilação, para que trechos de código só sejam compilados no Windows, e outros trechos só no Linux. Também não entraremos em muitos detalhes dessa diferença. Mas acostume-se com eles: você pode encontrar um desses de vez em quando.

Pronto. Nosso jogo funciona, e temos agora uma interface muito mais amigável para nosso jogo de Foge-foge.

Código final

O código final do jogo está em http://pastebin.com/U1M7VjFM

20.2 Resumindo

Neste capítulo, aprendemos:

- A entender o problema da redefinição de funções e structs que acontece quando nosso programa cresce em número de arquivos;

- A usar as diretivas de compilação `#ifdef` e `#ifndef` para resolver esses problemas.

Capítulo 21

Exercícios

Agora é sua vez de praticar!

21.1 Jogo Foge-foge

1) Suporte comandos tanto em letras maiúsculas quanto minúsculas.

2) Faça os fantasmas andarem na direção do herói. Tente otimizar o caminho deles para que eles cheguem o mais rápido possível ao herói.

3) Hoje nosso herói come fantasmas. Faça ele ter esse poder só se comer uma outra pílula. Assim que ele comê-la, ele terá esse poder de comer fantasmas por 20 turnos.

4) Faça os fantasmas não comerem as pílulas. Elas devem ser como uma parede para eles.

5) Transforme a função `explodepilula()` em uma função não recursiva. Use loops para isso. E, claro, deixe o código mais bonito possível.

6) Melhore a função `move()`. É possível fazer com que o `switch` não fique repetindo `m.matriz[][]` várias vezes.

21.2 Outros desafios

1) Desenhe a árvore de invocação para `fib(5)`. Você percebeu algo problemático?

2) Faça a função de `fib(5)` não precisar recalcular os mesmos valores mais de uma vez. Para isso, use um array para guardar os valores já calculados. Essa técnica é conhecida como **memorização**.

3) Escreva uma função que nos diga se uma frase é palíndroma. Frases palíndromas são aquelas que podem ser lidas de trás para frente. Por exemplo, *"Socorram-me, subi no ônibus em Marrocos!*

   é palíndroma.

Capítulo 22

O que fazer agora?

Se você chegou até aqui, parabéns! A caminhada foi longa, mas agora você já sabe o que precisa sobre programação e está pronto para começar a enfrentar desafios maiores.

Você ainda tem muito a estudar, como:

- **Orientação a Objetos (OO) e linguagens orientadas a objetos:** Java e C# dominam o mercado. Aprender o que é OO, polimorfismo, herança e encapsulamento são fundamentais para que você consiga lidar bem com essas linguagens.

- **Desenvolvimento para web:** não é necessário convencer alguém de que web é fundamental. Entender mais de HTML, CSS e Javascript do lado do cliente , e os frameworks para desenvolvimento do lado do servidor são importantes se você quer seguir nessa área.

- **Desenvolvimento mobile:** todo mundo tem celular. Android e iOS estão cada vez mais populares. Aprender Java e Objective-C/Swift são importantes, se você quer desenvolver aplicativos mobile.
- **Testes automatizados de código:** aqui testamos nossos programas de maneira manual. Ou seja, o executávamos, interagíamos com ele e víamos se ele funcionava. Isso não é nada profissional e impossível de se fazer em um sistema grande. Você precisa aprender a escrever programas que testam programas.
- **Boas práticas de código:** estudar Orientação a Objetos, padrões de projetos e arquitetura de software o ajudarão a escrever software pronto para o mundo real, e prontos para serem mantidos e evoluídos para sempre.

A vida do desenvolvedor de software é aprender sempre. Você não pode parar de buscar por fontes de informação. Nós, brasileiros, temos a sorte de ter uma comunidade de desenvolvimento bastante ativa. Participe de fóruns e eventos. Lá, você fará um ótimo *networking*, conhecerá pessoas, desenvolvedores mais experientes que você, e trocará experiências.

Não tenha medo de pesquisar. O Google é, com certeza, uma ferramenta primordial de trabalho. Também não se assuste com os erros do seu compilador. Leia-os com atenção e vá devagar, até encontrar o problema.

O caminho é longo, mas com certeza você já deu o primeiro passo. Não pare agora! E boa sorte com a sua caminhada!

Capítulo 23

Apêndice A: instalando o compilador

Se você usa Linux ou Mac, provavelmente já tem o GCC instalado. No Windows, o melhor a fazer é baixar o **Mingw/GCC**, disponível em http://sourceforge.net/projects/mingw. Instale os pacotes `base` e `c++`. Não esqueça também de colocar o diretório no `PATH`. No Windows, você faz isso em `Sistema -> Configurações Avançadas do Sistema -> Variáveis de Ambiente`.

Para testar sua instalação, o ideal seria compilar algum pequeno programa. Crie o arquivo a seguir e salve com o nome `teste.c`:

```
#include <stdio.h>

int main() {
```

```
    printf("Meu compilador funciona!");
}
```

Agora abra o terminal e digite `gcc teste.c -o teste.out`, no Linux/Mac; ou `gcc teste.c -o teste.exe`, no Windows. Em seguida, execute-o: no Linux/Mac, `./teste.out`; e no Windows, clicando duas vezes no arquivo. Você deverá ver a frase "Meu compilador funciona!".

Se ainda assim seu compilador não funcionar, o melhor a fazer é pesquisar o motivo no Google. Procurar por frases como "instalar GCC no Windows" ou o erro que está aparecendo na tela pode ajudar.

Capítulo 24

Apêndice B: códigos

24.1 Jogo da adivinhação

```
// incluindo as bibliotecas
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// definindo as constantes
#define NUMERO_DE_TENTATIVAS 3

int main() {

// imprimindo um cabeçalho bonito do jogo
printf("\n\n");
printf("          P  /_\\  P                              \n");
```

```
printf("         /_\\_|_|_/_\\                            \n");
printf("     n_n | ||. .|| | n_n        Bem-vindo ao     \n");
printf("     |_|_|nnnn nnnn|_|_|    Jogo de Adivinhação! \n");
printf("    |\" \"  |  |_|  |\"  \" |                    \n");
printf("    |_____| ' _ ' |_____|                        \n");
printf("          \\__|_|__/                             \n");
printf("\n\n");

// declarando variáveis que serão usadas mais à frente
int chute;
int acertou;
int nivel;
int totaldetentativas;

// definindo a quantidade de pontos inicial
double pontos = 1000;

// gerando um número secreto aleatório
srand(time(0));
int numerosecreto = rand() % 100;

// escolhendo o nível de dificuldade
printf("Qual o nível de dificuldade?\n");
printf("(1) Fácil (2) Médio (3) Difícil\n\n");
printf("Escolha: ");

scanf("%d", &nivel);

switch(nivel) {
    case 1:
        totaldetentativas = 20;
        break;
    case 2:
        totaldetentativas = 15;
        break;
    default:
        totaldetentativas = 6;
        break;
}
```

```
// loop principal do jogo
for(int i = 1; i <= totaldetentativas; i++) {

    printf("-> Tentativa %d de %d\n", i, totaldetentativas);

    printf("Chute um número: ");
    scanf("%d", &chute);

    // tratando chute de número negativo
    if(chute < 0) {
        printf("Você não pode chutar números negativos\n");
        i--;
        continue;
    }

    // verifica se acertou foi maior ou menor
    acertou = chute == numerosecreto;

    if(acertou) {
        break;
    } else if(chute > numerosecreto) {
        printf("\nSeu chute foi maior do que o número secreto!\n
                \n");
    } else {
        printf("\nSeu chute foi menor do que o número secreto!\n
                \n");
    }

    // calcula a quantidade de pontos
    double pontosperdidos = abs(chute - numerosecreto) / 2.0;
    pontos = pontos - pontosperdidos;
}

// imprimindo mensagem de vitória ou derrota
printf("\n");
if(acertou) {
    printf("             00000000000               \n");
    printf("         0000000000000000000           \n");
```

```
    printf("      000000  000000000  000000        \n");
    printf("    000000      00000      000000      \n");
    printf("  00000000  #   00000  #   00000000    \n");
    printf(" 0000000000    0000000    0000000000   \n");
    printf("0000000000000000000000000000000000000  \n");
    printf("0000000000000000000000000000000000000  \n");
    printf("0000  0000000000000000000000000  0000  \n");
    printf(" 0000  00000000000000000000000  0000   \n");
    printf("  0000   00000000000000000000  0000    \n");
    printf("    00000   000000000000000   0000     \n");
    printf("      000000   000000000   000000      \n");
    printf("         000000         000000         \n");
    printf("             000000000000              \n");
    printf("\nParabéns! Você acertou!\n");
    printf("Você fez %.2f pontos. Até a próxima!\n\n", pontos);
} else {

    printf("       \\|/ ____ \\|/    \n");
    printf("        @~/ ,. \\~@      \n");
    printf("       /_( \\__/ )_\\    \n");
    printf("          \\__U_/        \n");

    printf("\nVocê perdeu! Tente novamente!\n\n");
}


}
```

24.2 Jogo de forca

- **forca.c**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include "forca.h"

char palavrasecreta[TAMANHO_PALAVRA];
```

```
char chutes[26];
int chutesdados = 0;


int letraexiste(char letra) {

    for(int j = 0; j < strlen(palavrasecreta); j++) {
        if(letra == palavrasecreta[j]) {
            return 1;
        }
    }

    return 0;
}

int chuteserrados() {
    int erros = 0;

    for(int i = 0; i < chutesdados; i++) {

        if(!letraexiste(chutes[i])) {
            erros++;
        }
    }

    return erros;
}

int enforcou() {
    return chuteserrados() >= 5;
}

int ganhou() {
    for(int i = 0; i < strlen(palavrasecreta); i++) {
        if(!jachutou(palavrasecreta[i])) {
            return 0;
        }
    }
```

```
    return 1;
}


void abertura() {
    printf("/****************/\n");
    printf("/ Jogo de Forca */\n");
    printf("/****************/\n\n");
}

void chuta() {
    char chute;
    printf("Qual letra? ");
    scanf(" %c", &chute);

    if(letraexiste(chute)) {
        printf("Você acertou: a palavra tem a letra %c\n\n",
                chute);
    } else {
        printf("\nVocê errou: a palavra NÃO tem a letra %c\n\n",
                chute);
    }

    chutes[chutesdados] = chute;
    chutesdados++;
}

int jachutou(char letra) {
    int achou = 0;
    for(int j = 0; j < chutesdados; j++) {
        if(chutes[j] == letra) {
            achou = 1;
            break;
        }
    }

    return achou;
}
```

```
void desenhaforca() {

    int erros = chuteserrados();

    printf("  _______       \n");
    printf(" |/      |      \n");
    printf(" |      %c%c%c  \n", (erros>=1?'(':' '),
        (erros>=1?'_':' '), (erros>=1?')':' '));
    printf(" |      %c%c%c  \n", (erros>=3?'\\':' '),
        (erros>=2?'|':' '), (erros>=3?'/': ' '));
    printf(" |       %c     \n", (erros>=2?'|':' '));
    printf(" |      %c %c   \n", (erros>=4?'/':' '),
        (erros>=4?'\\':' '));
    printf(" |              \n");
    printf("_|___           \n");
    printf("\n\n");

    for(int i = 0; i < strlen(palavrasecreta); i++) {

        if(jachutou(palavrasecreta[i])) {
            printf("%c ", palavrasecreta[i]);
        } else {
            printf("_ ");
        }

    }
    printf("\n");

}

void escolhepalavra() {
    FILE* f;

    f = fopen("palavras.txt", "r");
    if(f == 0) {
        printf("Banco de dados de palavras não disponível\n\n");
        exit(1);
    }
```

```
    int qtddepalavras;
    fscanf(f, "%d", &qtddepalavras);

    srand(time(0));
    int randomico = rand() % qtddepalavras;

    for(int i = 0; i <= randomico; i++) {
        fscanf(f, "%s", palavrasecreta);
    }

    fclose(f);
}


void adicionapalavra() {
    char quer;

    printf("Você deseja adicionar uma nova palavra no
            jogo (S/N)?");
    scanf(" %c", &quer);

    if(quer == 'S') {
        char novapalavra[TAMANHO_PALAVRA];

        printf("Digite a nova palavra, em letras maiúsculas: ");
        scanf("%s", novapalavra);

        FILE* f;

        f = fopen("palavras.txt", "r+");
        if(f == 0) {
            printf("Banco de dados de palavras não
                    disponível\n\n");
            exit(1);
        }

        int qtd;
        fscanf(f, "%d", &qtd);
        qtd++;
```

```
        fseek(f, 0, SEEK_SET);
        fprintf(f, "%d", qtd);

        fseek(f, 0, SEEK_END);
        fprintf(f, "\n%s", novapalavra);

        fclose(f);

    }

}

int main() {

    abertura();
    escolhepalavra();

    do {

        desenhaforca();
        chuta();

    } while (!ganhou() && !enforcou());

    if(ganhou()) {
        printf("\nParabéns, você ganhou!\n\n");

        printf("       ___________      \n");
        printf("      '._==_==_=_.'     \n");
        printf("      .-\\:      /-.    \n");
        printf("     | (|:.     |) |    \n");
        printf("      '-|:.     |-'     \n");
        printf("        \\::.    /      \n");
        printf("         '::. .'        \n");
        printf("           ) (          \n");
        printf("         _.' '._        \n");
        printf("        '-------'       \n\n");

    } else {
```

```
        printf("\nPuxa, você foi enforcado!\n");
        printf("A palavra era **%s**\n\n", palavrasecreta);

        printf("    _______________         \n");
        printf("   /               \\      \n");
        printf("  /                 \\     \n");
        printf("//                   \\/\\  \n");
        printf("\\|   XXXX     XXXX   | /   \n");
        printf(" |   XXXX     XXXX   |/     \n");
        printf(" |   XXX       XXX   |      \n");
        printf(" |                   |      \n");
        printf(" \\__     XXX      __/     \n");
        printf("   |\\    XXX     /|       \n");
        printf("   | |           | |       \n");
        printf("   | I I I I I I I |       \n");
        printf("   |  I I I I I I  |       \n");
        printf("   \\_            _/       \n");
        printf("     \\_        _/         \n");
        printf("       \\_______/          \n");
    }

    adicionapalavra();
}
```

- **force.h**

```
#define TAMANHO_PALAVRA 20

int letraexiste(char letra);
int chuteserrados();
int enforcou();
void abertura();
void chuta();
int jachutou(char letra);
int ganhou();
void desenhaforca();
void escolhepalavra();
void adicionapalavra();
```

24.3 Foge-foge

- **fogefoge.c**

```
#include <stdio.h>
#include <stdlib.h>
#include "time.h"
#include "fogefoge.h"
#include "mapa.h"
#include "ui.h"

MAPA m;
POSICAO heroi;
int tempilula = 0;

int acabou() {
    POSICAO pos;

    int perdeu = !encontramapa(&m, &pos, HEROI);
    int ganhou = !encontramapa(&m, &pos, FANTASMA);

    return ganhou || perdeu;

}

int ehdirecao(char direcao) {
    return
        direcao == ESQUERDA ||
        direcao == CIMA ||
        direcao == BAIXO ||
        direcao == DIREITA;
}

void move(char direcao) {

    int proximox = heroi.x;
    int proximoy = heroi.y;

    switch(direcao) {
        case ESQUERDA:
```

```
            proximoy--;
            break;
        case CIMA:
            proximox--;
            break;
        case BAIXO:
            proximox++;
            break;
        case DIREITA:
            proximoy++;
            break;
    }

    if(!podeandar(&m, HEROI, proximox, proximoy))
        return;

    if(ehpersonagem(&m, PILULA, proximox, proximoy)) {
        tempilula=1;
    }

    andanomapa(&m, heroi.x, heroi.y, proximox, proximoy);
    heroi.x = proximox;
    heroi.y = proximoy;
}

int praondefantasmavai(int xatual, int yatual,
    int* xdestino, int* ydestino) {

    int opcoes[4][2] = {
        { xatual   , yatual+1 },
        { xatual+1 , yatual   },
        { xatual   , yatual-1 },
        { xatual-1 , yatual   }
    };

    srand(time(0));
    for(int i = 0; i < 10; i++) {
        int posicao = rand() % 4;
```

```
        if(podeandar(&m, FANTASMA, opcoes[posicao][0],
           opcoes[posicao][1])) {
            *xdestino = opcoes[posicao][0];
            *ydestino = opcoes[posicao][1];
            return 1;
        }
    }

    return 0;
}

void fantasmas() {
    MAPA copia;

    copiamapa(&copia, &m);

    for(int i = 0; i < copia.linhas; i++) {
        for(int j = 0; j < copia.colunas; j++) {
            if(copia.matriz[i][j] == FANTASMA)

                int xdestino;
                int ydestino;

                int encontrou = praondefantasmavai(i, j,
                                                    &xdestino,
                                                    &ydestino);

                if(encontrou) {
                    andanomapa(&m, i, j, xdestino, ydestino);
                }
            }
        }
    }

    liberamapa(&copia);
}

void explodepilula2(int x, int y, int somax, int somay, int qtd){
```

```
    if(qtd == 0) return;

    int novox = x+somax;
    int novoy = y+somay;

    if(!ehvalida(&m, novox, novoy)) return;
    if(ehparede(&m, novox, novoy)) return;

    m.matriz[novox][novoy] = VAZIO;
    explodepilula2(novox, novoy, somax, somay, qtd-1);
}

void explodepilula() {
    explodepilula2(heroi.x, heroi.y, 0, 1, 3);
    explodepilula2(heroi.x, heroi.y, 0, -1, 3);
    explodepilula2(heroi.x, heroi.y, 1, 0, 3);
    explodepilula2(heroi.x, heroi.y, -1, 0, 3);
}

int main() {

    lemapa(&m);
    encontramapa(&m, &heroi, HEROI);

    do {
        printf("Pílula: %s\n", (tempilula ? "SIM" : "NÃO"));
        imprimemapa(&m);

        char comando;
        scanf(" %c", &comando);

        if(ehdirecao(comando)) move(comando);
        if(comando == BOMBA) explodepilula();

        fantasmas();

    } while (!acabou());

    liberamapa(&m);
```

```
}
```

- **fogefoge.h**

```
#ifndef _FOGEFOGE_H_
#define _FOGEFOGE_H_

#define CIMA 'w'
#define BAIXO 's'
#define DIREITA 'd'
#define ESQUERDA 'a'
#define BOMBA 'b'

int acabou();
void move(char direcao);

int ehdirecao(char direcao);
void fantasmas();
void explodepilula();
void explodepilula2(int x, int y, int somax, int somay, int qtd);


#endif
```

- **mapa.c**

```
#include <stdio.h>
#include <stdlib.h>
#include "mapa.h"
#include <string.h>

void lemapa(MAPA* m) {
    FILE* f;
    f = fopen("mapa.txt", "r");
    if(f == 0) {
        printf("Erro na leitura do mapa");
        exit(1);
    }
```

```
    fscanf(f, "%d %d", &(m->linhas), &(m->colunas));
    alocamapa(m);

    for(int i = 0; i < m->linhas; i++) {
        fscanf(f, "%s", m->matriz[i]);
    }

    fclose(f);
}

void alocamapa(MAPA* m) {
    m->matriz = malloc(sizeof(char*) * m->linhas);

    for(int i = 0; i < m->linhas; i++) {
        m->matriz[i] = malloc(sizeof(char) * m->colunas + 1);
    }
}

void copiamapa(MAPA* destino, MAPA* origem) {
    destino->linhas = origem->linhas;
    destino->colunas = origem->colunas;
    alocamapa(destino);
    for(int i = 0; i < origem->linhas; i++) {
        strcpy(destino->matriz[i], origem->matriz[i]);
    }
}


void liberamapa(MAPA* m) {
    for(int i = 0; i < m->linhas; i++) {
        free(m->matriz[i]);
    }

    free(m->matriz);
}


int encontramapa(MAPA* m, POSICAO* p, char c) {
```

```
    for(int i = 0; i < m->linhas; i++) {
        for(int j = 0; j < m->colunas; j++) {
            if(m->matriz[i][j] == c) {
                p->x = i;
                p->y = j;
                return 1;
            }
        }
    }

    return 0;
}

int podeandar(MAPA* m, char personagem, int x, int y) {
    return
        ehvalida(m, x, y) &&
        !ehparede(m, x, y) &&
        !ehpersonagem(m, personagem, x, y);
}

int ehvalida(MAPA* m, int x, int y) {
    if(x >= m->linhas)
        return 0;
    if(y >= m->colunas)
        return 0;

    return 1;
}

int ehpersonagem(MAPA* m, char personagem, int x, int y) {
    return
        m->matriz[x][y] == personagem;
}

int ehparede(MAPA* m, int x, int y) {
    return
        m->matriz[x][y] == PAREDE_VERTICAL ||
        m->matriz[x][y] == PAREDE_HORIZONTAL;
}
```

```
void andanomapa(MAPA* m, int xorigem, int yorigem,
    int xdestino, int ydestino) {

    char personagem = m->matriz[xorigem][yorigem];
    m->matriz[xdestino][ydestino] = personagem;
    m->matriz[xorigem][yorigem] = VAZIO;

}
```

- **mapa.h**

```
#ifndef _MAPA_H_
#define _MAPA_H_


#define HEROI '@'
#define VAZIO '.'
#define PAREDE_VERTICAL '|'
#define PAREDE_HORIZONTAL '-'
#define FANTASMA 'F'
#define PILULA 'P'

struct mapa {
    char** matriz;
    int linhas;
    int colunas;
};

typedef struct mapa MAPA;

void alocamapa(MAPA* m);
void lemapa(MAPA* m);
void liberamapa(MAPA* m);

struct posicao {
    int x;
    int y;
```

```
};

typedef struct posicao POSICAO;

int encontramapa(MAPA* m, POSICAO* p, char c);

int ehvalida(MAPA* m, int x, int y);
int ehparede(MAPA* m, int x, int y);
int ehpersonagem(MAPA* m, char personagem, int x, int y);

void andanomapa(MAPA* m, int xorigem, int yorigem,
    int xdestino, int ydestino);

void copiamapa(MAPA* destino, MAPA* origem);

int podeandar(MAPA* m, char personagem, int x, int y);

#endif
```

- **ui.c**

```
#include <stdio.h>
#include "mapa.h"

char desenhoparede[4][7] = {
    {"......" },
    {"......" },
    {"......" },
    {"......" }
};

char desenhofantasma[4][7] = {
    {" .-.  " },
    {"| OO| " },
    {"|   | " },
    {"'^^^' " }
};

char desenhoheroi[4][7] = {
```

```
    {" .--. "  },
    {"/ _.-'"  },
    {"\\  '-." },
    {" '--' "  }
};

char desenhopilula[4][7] = {
    {"      "},
    {" .-.  "},
    {" '-'  "},
    {"      "}
};

char desenhovazio[4][7] = {
    {"      "},
    {"      "},
    {"      "},
    {"      "}
};

void imprimeparte(char desenho[4][7], int parte) {
    printf("%s", desenho[parte]);
}

void imprimemapa(MAPA* m) {
    for(int i = 0; i < m->linhas; i++) {

        for(int parte = 0; parte < 4; parte++) {
            for(int j = 0; j < m->colunas; j++) {

                switch(m->matriz[i][j]) {
                    case FANTASMA:
                        imprimeparte(desenhofantasma, parte);
                        break;
                    case HEROI:
                        imprimeparte(desenhoheroi, parte);
                        break;
                    case PILULA:
                        imprimeparte(desenhopilula, parte);
```

```
                        break;
                    case PAREDE_VERTICAL:
                    case PAREDE_HORIZONTAL:
                        imprimeparte(desenhoparede, parte);
                        break;
                    case VAZIO:
                        imprimeparte(desenhovazio, parte);
                        break;
                }

            }
            printf("\n");
        }

    }
}
```

- **ui.h**

```
#ifndef _UI_H_
#define _UI_H_

#include "mapa.h"

void imprimeparte(char desenho[4][7], int parte);
void imprimemapa(MAPA* m);

#endif
```